U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE

TRANSMITTAL LETTER TO THE UNITED STATES DESIGNATED/ELECTED OFFICE (DO/EO/US) CONCERNING A FILING UNDER 35 U.S.C. 371

ATTORNEY'S DOCKET NUMBER
1454.1120

09/980489

INTERNATIONAL APPLICATION NO.	INTERNATIONAL FILING DATE	PRIORITY DATE CLAIMED
PCT/DE00/01001	3 April 2000	2 June 1999

TITLE OF INVENTION
METHOD AND ARRANGEMENT FOR DETERMINING A TOTAL ERROR DESCRIPTION OF AT LEAST ONE PART OF A COMPUTER PROGRAMME AND COMPUTER PROGRAMME PRODUCT AND COMPUTER-READABLE STORAGE MEDIUM

APPLICANT(S) FOR DO/EO/US
Peter LIGGESMEYER

Applicant herewith submits to the United States Designated/Elected Office (DO/EO/US) the following items and other information:

1. ☒ This is a FIRST submission of items concerning a filing under 35 U.S.C. 371.
2. ☒ This is an express request to immediately begin national examination procedures (35 U.S.C. 371(f)).
3. ☒ The US has been elected by the expiration of 19 months from the priority date (PCT Article 31).
4. ☒ A copy of the International Application as filed (35 U.S.C. 371(c)(2))
   a. ☒ is transmitted herewith (required only if not transmitted by the International Bureau).
   b. ☐ has been transmitted by the International Bureau.
   c. ☐ is not required, as the application was filed in the United States Receiving Office RO/US).
5. ☒ A translation of the International Application into English (35 U.S.C. 371(c)(2)).
6. ☐ Amendments to the claims of the International Application under PCT Article 19 (35 U.S.C. 371(c)(3))
   a. ☐ are transmitted herewith (required only if not transmitted by the International Bureau).
   b. ☐ have been transmitted by the International Bureau.
   c. ☐ is not required, as the application was filed in the United States Receiving Office (RO/US)
7. ☐ A translation of the amendments to the claims under PCT Article 19 (35 U.S.C. 371(c)(3)).
8. ☒ An oath or declaration of the inventor (35 U.S.C. 371(c)(4)).
9. ☒ A translation of the Annexes to the International Preliminary Examination Report under PCT Article 36 (35 U.S.C. 371(c)(5)).

Items 10-15 below concern document(s) or information included:

10. ☒ An Information Disclosure Statement Under 37 CFR 1.97 and 1.98.
11. ☒ An assignment document for recording.
    Please mail the recorded assignment document to:
    a. ☒ the person whose signature, name & address appears at the bottom of this document.
    b. ☐ the following:
12. ☒ A preliminary amendment.
13. ☒ A substitute specification
14. ☐ A change of power of attorney and/or address letter.
15. ☒ Other items or information:

PCT EASY forms filed with International Application; copies of cover page of International Application as published, International Search Report, and International Preliminary Examination Report; and two CD-R discs in IBM-PC format with MS-Windows operating system compatibility, each storing the file PROGRA~6.TXT (Program-Listing.txt in Windows) of 130,121 bytes, created November 29, 2001.

09/980489

☒ The U.S. National Fee (35 U.S.C. 371(c)(1)) and other fees as follows:

CLAIMS	(1) FOR	(2) NUMBER FILED	(3) NUMBER EXTRA	(4) RATE	(5) CALCULATIONS
	TOTAL CLAIMS	15 -20=	0	x $ 18.00	0.00
	INDEPENDENT CLAIMS	3 -3=	0	x $ 84.00	0.00
	MULTIPLE DEPENDENT CLAIM(S) (if applicable)			+$280.00	0.00
	BASIC NATIONAL FEE (37 CFR 1.492(a)(1)-(4): ☐ Neither international preliminary examination fee (37 CFR 1.482) nor international search fee (37 CFR 1.445(a)(2)) paid to USPTO .....$1,040 ☒ International preliminary examination fee (37 C.F.R. 1.482) not paid to USPTO but International Search Report prepared by the EPO or JPO.....$ 890 ☐ International preliminary examination fee (37 C.F.R. 1.482) not paid to USPTO but international search fee (37 C.F.R. 1.445(a)(2) paid to USPTO...$ 740 ☐ International preliminary examination fee paid to USPTO (37 CFR 1.482) but all claims did not satisfy provision of PCT Article 33(1)-(4).....$ 710 ☐ International preliminary examination fee paid to USPTO (37 CFR 1.482) and all claims satisfied provisions of PCT Article 33(2) to (4) .....$ 100				890.00
	Surcharge of $130 for furnishing the National fee or oath or declaration later than ☐ 20 ☐ 30 mos. from the earliest claimed priority date (37 CFR 1.482(e)).				0.00
			TOTAL OF ABOVE CALCULATIONS		890.00
	Reduction by 1/2 for filing by small entity, if applicable. Affidavit must be filed also. (Note 37 CFR 1.9, 1.27, 1.28.)				
			SUBTOTAL		890.00
	Processing fee of $130 for furnishing the English Translation later than [ ] 20 [ ] 30 mos. from the earliest claimed priority date (37 CFR 1.482(f)).				
			TOTAL NATIONAL FEE		890.00
	Fee for recording the enclosed assignment (37 CFR 1.21(h)).				+ 40.00
			TOTAL FEES ENCLOSED		930.00

a. ☒ A check in the amount of $ 930.00 to cover the above fees is enclosed.
b. ☐ Please charge my Deposit Account No. 19-3935 in the Amount of $ to cover the above fees. A duplicate copy of this sheet is enclosed.
c. ☒ The Commissioner is hereby authorized to charge any additional fees which may be required, or credit any overpayment to Deposit Account No. 19-3935. A duplicate copy of this sheet is enclosed.

21171
PATENT TRADEMARK OFFICE

SUBMITTED BY: STAAS & HALSEY LLP

Type Name	Richard A. Gollhofer	Reg. No.	31,106
Signature	Richard A. Gollhofer	Date	12/3/01

Docket No.: (NEW) 1454.1120

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re the Application of:

Peter LIGGESMEYER

Serial No. Group Art Unit: (NEW)

Confirmation No.

Filed: (concurrently) Examiner:

For: SYSTEM FOR DETERMINING A TOTAL ERROR DESCRIPTION OF AT LEAST ONE PART OF A COMPUTER PROGRAM (as amended)

PRELIMINARY AMENDMENT

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

Before examination of the above-identified application, please amend the application as follows:

IN THE TITLE:

Please DELETE the Title in its entirety and REPLACE with the following new Title.

--SYSTEM FOR DETERMINING A TOTAL ERROR DESCRIPTION OF AT LEAST ONE PART OF A COMPUTER PROGRAM--

IN THE ABSTRACT:

Please DELETE the Abstract in its entirety and substitute the attached new Abstract.

IN THE SPECIFICATION:

Please REPLACE the Specification in its entirety with the attached Substitute Specification wherein the program listing on pages 20-82 has been deleted and replaced by the program listing in the compact disc appendix submitted herewith.

IN THE CLAIMS:

Please CANCEL claims 1-16 without prejudice or disclaimer and ADD new claims 17-31 in accordance with the following:

17. (NEW) A method for using a computer to ascertain an overall fault description for at least one section of a computer program, comprising:

storing a section of the computer program;

ascertaining a control flow description for the section of the computer program, to describe a flow of control information in the section of the computer program;

ascertaining a data flow description for the section of the computer program, to describe a flow of data in the section of the computer program;

combining the control and data flow descriptions into a joint flow description for the section of the computer program;

selecting program elements from the section of the computer program;

storing a fault description for each reference element to describe possible faults in the reference element;

ascertaining an element fault description for each selected program element, based on the fault description associated with a corresponding reference element, to describe possible faults in the selected program element; and

ascertaining the overall fault description using the element fault descriptions, with a structure of the overall fault description taking into account a structure of the joint flow description.

18. (NEW) The method as claimed in claim 17, wherein the control flow description is a control flow graph.

19. (NEW) The method as claimed in claim 17, wherein the data flow description is a data flow graph.

20. (NEW) The method as claimed in claim 17,

wherein the fault description is a stored fault tree,

wherein the element fault description is ascertained as an element fault tree, and

wherein the overall fault description is ascertained as an overall fault tree.

21. (NEW) The method as claimed in claim 17, further comprising performing fault analysis in the section of the computer program using the overall fault description.

22. (NEW) The method as claimed in claim 17,

wherein the overall fault description is ascertained as an overall fault tree, and

wherein said method further comprises altering the overall fault tree in terms of prescribable boundary conditions.

23. (NEW) The method as claimed in claim 22, wherein said altering comprises adding a complementary fault tree.

24. (NEW) A system for ascertaining an overall fault description for at least one section of a computer program, comprising:

a storage unit to store the section of the computer program and fault descriptions for reference elements, each fault description describing possible faults in one of the reference elements; and

a processor, coupled to said storage unit,

to ascertain control and data flow descriptions for the section of the computer program, the control flow description describing a flow of control information in the section of the computer program and the data flow description describing a flow of data in the section of the computer program,

to combine the control and data flow descriptions into a joint flow description for the section of the computer program,

to select program elements from the section of the computer program,

to ascertain element fault descriptions for the program elements that have been selected, to describe possible faults in each program element based on the fault description associated with a corresponding reference element; and

to ascertain the overall fault description from the element fault descriptions, with a structure of the overall fault description taking into account a structure of the joint flow description.

25. (NEW) The system as claimed in claim 24, wherein said processor ascertains the control flow description as a control flow graph.

26. (NEW) The system as claimed in claim 24, wherein said processor ascertains the data flow description as a data flow graph.

27. (NEW) The system as claimed in claim 24,

wherein said storage unit stores each fault description as a fault tree, and

wherein said processor ascertains each element fault description as an element fault tree and the overall fault description as an overall fault tree.

28. (NEW) The system as claimed in claim 24, wherein said processor further performs fault analysis in the section of the computer program using the overall fault description.

29. (NEW) The system as claimed in claim 24, wherein said processor ascertains the overall fault description as an overall fault tree and alters the overall fault tree in terms of prescribable boundary conditions.

30. (NEW) The system as claimed in claim 29, wherein said processor alters the overall fault tree by adding a complementary fault tree.

31. (NEW) A computer-readable storage medium storing at least one program to control a computer to perform a method for ascertaining an overall fault description for at least one section of a computer program, said method comprising:

storing a section of the computer program;

ascertaining a control flow description for the section of the computer program, to describe a flow of control information in the section of the computer program;

ascertaining a data flow description for the section of the computer program, to describe a flow of data in the section of the computer program;

combining the control and data flow descriptions into a joint flow description for the section of the computer program;

selecting program elements from the section of the computer program;

storing a fault description for each reference element to describe possible faults in the reference element;

ascertaining an element fault description for each selected program element, based on the fault description associated with a corresponding reference element, to describe possible faults in the selected program element; and

ascertaining the overall fault description using the element fault descriptions, with a structure of the overall fault description taking into account a structure of the joint flow description.

REMARKS

This Preliminary Amendment is submitted to improve the form of the English translation as filed. It is respectfully requested that this Preliminary Amendment be entered in the above-referenced application.

In accordance with the foregoing, claims 1-16 have been canceled and claims 17-31 have been added. Thus, claims 17-31 are pending and under consideration.

A substitute specification is also being filed herewith. The substitute specification is accompanied by a marked-up copy of the original specification.

If there are any additional fees associated with filing of this Preliminary Amendment, please charge the same to our Deposit Account No. 19-3935.

Respectfully submitted,

STAAS & HALSEY LLP

Date: 12/3/01

By: Richard A. Gollhofer
Richard A. Gollhofer
Registration No. 31,106

700 Eleventh Street, NW, Suite 500
Washington, D.C. 20001
(202) 434-1500

SUBSTITUTE SPECIFICATION

TITLE OF THE INVENTION

SYSTEM FOR DETERMINING A TOTAL ERROR
DESCRIPTION OF AT LEAST ONE PART OF A COMPUTER PROGRAM

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application is based on and hereby claims priority to German Patent Application No. 19925239.4 filed on June 2, 2001, the contents of which are hereby incorporated by reference.

REFERENCE TO COMPUTER PROGRAM LISTING, COMPACT DISC APPENDIX

[0002] A compact disc is included herewith and incorporated by reference herein having thereon a computer program listing appendix in the ASCII uncompressed text format with ASCII carriage return, ASCII line feed and all control codes defined in ASCII, having computer compatibility with IBM PC/XT/AT or compatibles, having operating system compatibility with MS-Windows and including file PROGRA~6.TXT (Program-Listing.txt in Windows) of 130,121 bytes, created on November 29, 2001.

BACKGROUND OF THE INVENTION

[0003] The invention relates to a method and a system for ascertaining an overall fault description for at least one section of a computer program, and also to a computer product and a computer-readable storage medium.

[0004] Such a method and such a system are known from N. Leveson, "Safety Verification of ADA Programs Using Software Fault Trees", IEEE Software, July 1991, pages 48-59, which discloses the practice of using computers to ascertain an overall fault description in the form of an overall fault tree for a computer program. For the computer program, a control flow description is ascertained in the form of a control flow graph. For various program elements of the computer program, a stored fault description associated with a respective stored reference element is used to ascertain an element fault description. The fault description for a reference element describes possible faults in the respective reference element. The element fault descriptions in the form of element fault trees are used to ascertain the overall fault description, taking into account the control flow graph for the computer program.

[0005] The method and the system taught by Leveson have the following drawbacks, in particular. The overall fault tree ascertained is incomplete in terms of the faults examined and the causes thereof, and is therefore unreliable. Hence, this practice is not appropriate for use within the context of generating fault trees for a computer program for safety-critical applications. The individual fault trees associated with the reference elements are also incomplete and hence unreliable.

[0006] M. Weiser, "Program Slicing", in IEEE Transaction on Software Engineering, Vol. 10, No. 4, July 1984, pp. 352-357 provides an overview of "slicing". Slicing is the analysis carried out when searching for causes of incorrect action in a computer program. This procedure involves checking whether the incorrect action has been caused by an instruction currently under consideration. If this is not the case, the instructions which deliver data for or control the execution of the instruction are checked. This method is continued until no further operations exist, that is to say it gets to input data for the computer program. In slicing, "slices" are ascertained. A slice shows which instructions are affected in what way by a value under consideration. Below, the term slicing is always understood to mean backwardly directed slicing.

[0007] P. Liggesmeyer, Modultest und Modulverifikation – State of the Art, Mannheim, Vienna, Zurich: BI Wissenschaftsverlag, 1990 discloses the practice of ascertaining a control flow description and a data flow description for a computer program. In Liggesmeyer, this representation is used as an initial basis for "data-flow-oriented testing" of the computer program. The instructions (nodes) of the control flow graph are assigned data flow attributes (data flow description) which describe the nature of the data access operations contained in the instructions of the computer program. A distinction is drawn between write access operations and read access operations. Write access operations are referred to as definitions (def). Read access operations are referred to as a reference. If a read access operation takes place in a decision, this access operation is referred to as a predicative reference (p-use, predicate use). A read access operation during calculation of a value is referred to as a computational reference (c-use, computational use).

[0008] DIN 25424-1: Fehlerbaumanalysen; Methoden und Bildzeichen, September 1981, which has a title that can be translated "Fault Tree Analyses; Methods and Graphic Symbols", discloses principles relating to a fault tree. A fault tree is to be understood, as described in DIN 25424-1, to mean a structure which describes logical relationships between input variables for the fault tree which lead to a prescribed undesirable event.

[0009] In addition, DIN 25424-2: Fehlerbaumanalyse; Handrechenverfahren zur Auswertung eines Fehlerbaums, Berlin, Beuth Verlag GmbH, April 1990 which has a title that can be translated "Fault Tree Analysis; Manual Computation Methods for Evaluating a Fault Tree", discloses various methods for fault tree analysis.

SUMMARY OF THE INVENTION

[0010] The invention is based on the problem of ascertaining an overall fault description which is more reliable than ascertaining an overall fault tree in the manner known on the basis of the method taught by Leveson.

[0011] In a method for ascertaining an overall fault description for at least one section of a computer program, using a computer, at least the section of the computer program is stored. A control flow description and a data flow description are ascertained for the section of the computer program, and program elements are selected from the section of the computer program. For each selected program element, a stored fault description is used to ascertain an element fault description. The fault description is associated with a respective reference element. The element fault description describes possible faults in the respective program element. A fault description for a reference element describes possible faults in the respective reference element. The element fault descriptions are used to ascertain the overall fault description, which takes into account the control flow description and the data flow description.

[0012] A system for ascertaining an overall fault description for at least one section of a computer program has a processor which is set up such that the following method steps can be carried out:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,

- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

[0013] A computer program product comprises a computer-readable storage medium on which a program is stored which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,
- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

[0014] A computer-readable storage medium stores a program which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,

- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

[0015] The invention now makes it possible to ascertain a reliable overall fault description, which takes into account the peculiarities of a computer program, for a computer program or a section thereof. Since the overall fault description ascertained is much more reliable than the overall fault description which can be ascertained on the basis of the method taught by Leveson, the invention is also suitable for safety-critical applications, i.e. in particular for ascertaining an overall fault description for a safety-critical computer program.

[0016] The control flow description and/or the data flow description may be in the form of a control flow graph or of a data flow graph, respectively.

[0017] The fault description may be in the form of a stored fault tree, and the element fault description can be ascertained as an element fault tree. In this case, the overall fault description can be ascertained as an overall fault tree. This development permits standardized representation of a fault description, which makes it much simpler for a user of the fault description to analyze same.

[0018] In one development, the overall fault description can be used for fault analysis in the section of the computer program. This development has the advantage, in particular, that automated, reliable fault analysis becomes possible, and if the fault descriptions are in the form of fault trees the fault description can even be analyzed in a manner "normalized" in accordance with the fault tree analysis methods.

[0019] In another refinement, the overall fault description is ascertained as an overall fault tree, and the overall fault tree is altered in terms of prescribable boundary conditions. The alteration can be made by adding a complementary fault tree.

BRIEF DESCRIPTION OF THE DRAWINGS

[0020] These and other objects and advantages of the present invention will become more apparent and more readily appreciated from the following description of the exemplary embodiment of the invention as explained in more detail below and as illustrated in the figures., in which:

[0021] Figure 1 is a block diagram of a computer used to carry out the method in accordance with the exemplary embodiment;

[0022] Figure 2 is a flowchart showing the individual method steps of the method in accordance with the exemplary embodiment;

[0023] Figure 3 is a general fault tree, as formed basically for a reference element;

[0024] Figures 4a to 4c are flow diagrams for a control flow graph (Fig. 4a), a slice (Fig. 4b) and a fault tree (Fig. 4c) for an instruction sequence as a reference element of a computer program;

[0025] Figures 5a to 5c are flow diagrams for a control flow graph (Fig. 5a), a slice (Fig. 5b) and a fault tree (Fig. 5c) for a selection sequence as a reference element of a computer program;

[0026] Figures 6a to 6c are flow diagrams for a control flow graph (Fig. 6a), a slice (Fig. 6b) and a fault tree (Fig. 6c) for a loop as a reference element of a computer program;

[0027] Figure 7 is a control flow graph with a data flow graph for a computer program in accordance with the exemplary embodiment;

[0028] Figures 8a and 8b are flow diagrams for a slice for the output of the variable max (Fig. 8a) and a slice for the variable avr (Fig. 8b) for the program in accordance with the exemplary embodiment;

[0029] Figure 9 is a flow diagram for the slice for the variable avr, in which a structure of the loop from the program in accordance with the exemplary embodiment is highlighted;

[0030] Figure 10 is a flow diagram for a fault tree for the assumption that the variable avr is incorrect;

[0031] Figure 11 is a flow diagram for the overall fault tree based on Fig. 10, where redundant events from the overall fault tree based on Fig. 10 have been combined into one event.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

[0032] Reference will now be made in detail to the preferred embodiments of the present invention, examples of which are illustrated in the accompanying drawings, wherein like reference numerals refer to like elements throughout.

[0033] Figure 1 shows a computer 100 used to carry out the method described below. The computer 100 has a processor 101 which is connected to a memory 102 via a bus 103. The bus 103 also has an input/output interface 106 connected to it.

[0034] The memory 102 stores a computer program 104 for which an overall fault description is ascertained in the manner described below. In addition, the memory 102 stores a program 105 which implements the method described below. The memory also stores fault descriptions 115 for different reference elements of a computer program. A fault description for a reference element describes possible faults in the respective reference element. Various reference elements and fault descriptions associated with the reference elements are explained in detail below.

[0035] The input/output interface 106 has a keyboard 108 connected to it via a first connection 107. A second connection 109 is used to connect the input/output interface 106 to a computer mouse 110, and a third connection 111 is used to connect the input/output interface 106 to a screen 112 on which the overall fault description ascertained for the computer program 104 is displayed. A fourth connection 113 is used to connect the input/output interface 106 to an external storage medium 114.

[0036] Figure 2 shows a block diagram illustrating the procedure in accordance with the exemplary embodiment described below.

[0037] The stored computer program 104 is used to ascertain a control flow graph 201 and a data flow graph 202 for the computer program 104.

[0038] Individual program elements are selected from the computer program (step 203). For each program element selected, a stored fault description associated with a reference element corresponding to the selected program element is used to ascertain an element fault description (step 204). The element fault description describes possible faults in the respective selected program element.

[0039] On the basis of a fault event in the computer program (undesirable event), which fault event is prescribed by a user and needs to be examined, in a final step (step 205) an overall fault description for the computer program is ascertained, for the fault instance to be examined, from the element fault descriptions, taking into account the control flow graph and the data flow graph. The overall fault tree ascertained is displayed to the user on the screen 112.

[0040] Figure 3 shows the basic procedure for creating a fault tree, as used in the initial example in order to form the fault trees described below for the reference elements.

[0041] For an event 301 selected by a user, it is necessary to ascertain how the selected incorrect event can arise. In a computer program, incorrect output of a variable, as a selected

incorrect event (undesirable event) 301, can be caused by a control flow fault 303 and/or a data fault 304 (INCLUSIVE-OR function 302).

[0042] A control flow fault 303 is to be understood to mean incorrect control of the processing of the respective variable.

[0043] The data flow fault 304 is to be understood to mean a fault which arises during processing as a result of incorrect data. The data flow fault 304 may originate in the processing step currently under consideration (block 306) and/or it may already have been present and may be maintained only by fault propagation (block 307) (INCLUSIVE-OR function 305).

[0044] On the basis of these considerations, the appropriate fault tree, a slice describing the instruction and a control flow graph are respectively illustrated below for the following elements of a computer program:

- an instruction sequence,
- a selection element,
- a loop element.

Instruction sequence

[0045] The instruction sequence 401 comprises the three instructions shown in Fig. 4a. In a first instruction 402, a first variable j is assigned the value 3 (j: = 3). A second instruction 403 assigns a second variable k the value 2 (k: = 2). A third instruction 404 forms a sum using the first variable and the second variable (i: = j + k).

[0046] In accordance with the practice disclosed in Weiser, a slice 410 is formed for this instruction sequence 401, as shown in Fig. 4b. The first instruction 402 and the second instruction 403 both affect the third instruction 404, which is illustrated by two arrows 411, 412 in the slice 410.

[0047] For the control flow graph 401, the fault tree 420 shown in Fig. 4c is obtained for the following prescribed undesirable event 421: "Variable i is incorrect after the third instruction".

[0048] The incorrect event 421 may have been produced by a fault in the third instruction 404 under consideration itself, if the data up to this instruction step were correct (element 422 in Fig. 4c). The incorrect event 421 may also be caused by corrupt input data for the third instruction, however, i.e. as a result of INCLUSIVE-ORing 424 the events that the second variable k was incorrect after the second instruction (element 425) and/or that the first variable j was incorrect after the first instruction 402 (element 426). The result of the first INCLUSIVE-ORed function

424 is INCLUSIVE-ORed with the event that the third instruction is incorrect (INCLUSIVE-OR function 423).

Selection element

[0049] With a selection element as reference element, it is necessary to consider possibilities of fault in the data flows and in the control flows within the computer program.

[0050] Figures 5a to 5c show a control flow graph 501 (cf. Fig. 5a), a slice 520 (cf. Fig. 5b) and a fault tree 540 (cf. Fig. 5c) for an If-Then-Else instruction as a selection element. The control flow graph 501 comprises the following six instructions:

- a first instruction 502, which assigns a first variable j the value 3 (j: = 3),
- a second instruction 503, which assigns a second variable k a prescribable value (k: = ...),
- a third instruction 504, which checks whether the second variable k has a value greater than 0; if the value of the second variable is greater than 0, the instruction branches to a fourth instruction 505, otherwise it branches to a fifth instruction 506,
- the fourth instruction 505, which assigns a third variable i the value of the second variable k (i: = k),
- a fifth instruction 506, which assigns the third variable i the value of the second variable k with a negative arithmetic sign (i: = -k),
- a sixth instruction 507, which processes the third variable i further in an arbitrary manner.

[0051] For the control flow graph 501 shown in Fig. 5a, the slice 520 shown in Fig. 5b is obtained for the selection element. Solid edges in the slice 520 show a data dependency between the different instructions. Dashed edges indicate control dependencies between the appropriate instructions.

[0052] The following definitions apply for the two edge types:

- dashed edges, referred to as control edges below, are directed from instructions which contain a predicative reference (failure constructs, loop control) to the directly controlled instructions, i.e. to those instructions which are executed only if the predicate has a particular value. Control edges are drawn only between the controlling instruction and directly interleaved instructions. If a controlled block contains a further interleaved control level, no control edges crossing more than one level are drawn. Since a control relationship is transitive, this indirect control can be inferred from the slice by utilizing the transitivity.

- solid edges, referred to as data flow edges below, are directed from instructions in which a variable is defined to instructions in which this variable is referenced. The variable under consideration cannot be defined again between the definition and the reference. This is referred to as a definition-free path for the variable under consideration.

[0053] The slice is ascertained by searching the control flow graph, counter to the edge direction, for a definition of the variable under consideration starting from the instruction containing the variable under consideration, for which the undesirable event is prescribed. If computational references exist for the definition, the method is continued recursively until no further additional nodes are found. The dependencies found in this way between instructions are data dependencies. If a node under consideration is contained in a block whose execution is controlled directly by a decision, this represents a control dependency. For the predicative references of the variables involved in the decision, nodes with appropriate definitions - that is to say data flow dependencies - are recursively sought which have other control dependencies.

[0054] Figure 5b shows the failure element's associated slice 520 with corresponding control edges and data flow edges.

[0055] Figure 5c shows the fault tree 540 for the prescribed event "the third variable i is incorrect before the 6th instruction" 541.

[0056] The following events result in the incorrect event 541 when INCLUSIVE-ORed 542:

- ANDing 543 the events that the decision in accordance with the third instruction 504 is true (element 544) and a result of INCLUSIVE-ORing 545 the events that the fourth instruction 505 is incorrect (element 546) and/or the first variable j is incorrect after the first instruction 502 (element 547);
- ANDing 550 the events that the decision in accordance with the third instruction 504 is false (element 551) and a result of INCLUSIVE-ORing 552 the events that the fifth instruction is incorrect (element 553) and/or that the first variable j is incorrect after the first instruction 502 (element 554);
- INCLUSIVE-ORing 560 the following events: the decision in accordance with the third instruction 504 is incorrect (element 561) and/or the second variable k is incorrect after the second instruction 503 (element 562).

Multiple selection element

[0057] A multiple selection element as reference element can be handled in accordance with the scheme described above by breaking down the multiple selection into a cascade of two-way selection elements processed in accordance with the procedure above, in order thus to ascertain a fault tree for a multiple selection element.

Loop

[0058] Figures 6a to 6c show a fault tree 601 (cf. Fig. 6a), the corresponding slice 620 (cf. Fig. 6b) and the associated fault tree 640 (cf. Fig. 6c) for the reference element of a loop. The control flow graph 601 for a loop element comprises the following seven instructions:

- a first instruction 602, which assigns a first variable i the value 0 (i: = 0),
- a second instruction 603, which assigns a second variable j an unspecified value (j: = ...),
- a third instruction 604, which prescribes a further unspecified value for a third variable k (k: = ...),
- a fourth instruction 605, which, as a loop instruction, specifies a condition that a fifth instruction and a sixth instruction are executed until the value of the second variable is j > 0 (WHILE j > 0 DO),
- a fifth instruction 606, which assigns the first variable i a value which is obtained from the sum of the previous value of the first variable and the product of the second variable and the third variable (i: = i + k * j),
- a sixth instruction 607, which assigns the second variable j a value which is obtained by decreasing the original value of the second variable j by the value 1 (j: = j - 1),
- a seventh instruction 608, which processes the first variable i further in a prescribable manner (...: = i ...).

[0059] Figure 6b shows the corresponding slice 620 for the control flow graph 601 shown in Fig. 6a with associated control flow edges and data flow edges. The fault tree 640 shown in Fig. 6c is formed for the prescribed event 641 that the "first variable i is incorrect before the seventh instruction".

[0060] The fault tree 640 is obtained by INCLUSIVE-ORing 642 the following four events:

- a first event 643, which describes a situation in which the first variable i is incorrect after the first instruction 602,

- ANDing 644 the events that the loop body has been passed through at least twice (element 645) and the event that the sixth instruction 607 is incorrect (646),
- ANDing 650 the event that the loop body has been executed at least once (element 651) and INCLUSIVE-ORing 652 of the following four events:

  a) the fifth instruction 606 is incorrect (element 653),

  b) the first variable i is incorrect after the first instruction (element 654),

  c) the second variable j is incorrect after the second instruction (element 655),

  d) the third variable k is incorrect after the third instruction (element 656),
- INCLUSIVE-ORing 660 the following three events:

  e) the decision in accordance with the fourth instruction 605 is incorrect (element 661),

  f) the second variable j is incorrect after the second instruction 603 (element 662),

  g) ANDing 663 the events that the sixth instruction is incorrect (element 664) and the event that the loop body has been passed through at least once (element 665).

[0061] The fault trees described above, which are associated with the individual reference elements, are stored in the memory 102 as fault trees 115.

[0062] Figure 7 shows a control flow graph 700 for the following computer program:

```
input (n);
input (a);
max:=0;
sum:=0;
i:=2;
WHILE i =n DO
        IF max < a
        THEN max:= a
        sum:= sum + a
        i:= i + 1
avr:= sum/n;
output (max);
output (avr);
```

[0063] For the control flow graph 700 comprising 13 instructions (reference symbols 1, 2, 3, ..., 13) which is shown in Fig. 7, Fig. 8a shows the associated slice 800 for the variable max and Fig. 8b shows the associated slice 810 for the variable avr. The numbering of the individual instructions in the slices corresponds to the numbering of the individual instructions in the control flow graph 700 from Fig. 7.

[0064] Figure 9 shows the slice 900 for the variable avr, as shown in Fig. 8b. The structure of the loop element contained in the program shown above is highlighted in bold. This structure corresponds to the slice shown in Fig. 6b for a loop element.

[0065] An overall fault tree 1000 for the computer program shown above is shown in Fig. 10. The overall fault tree for the computer program is produced by instantiating the appropriate fault tree associated with the reference element which corresponds to the selected program element.

[0066] By starting from the prescribed undesirable event and working backward, the overall fault tree 1000 is thus ascertained using the fault trees associated with the reference elements.

[0067] Figure 10 contains the fault tree 1000 relating to the event that "the variable avr is incorrect before the thirteenth instruction" (element 1001). The variable avr may be incorrect before the thirteenth instruction 13 on account of at least one of the following three events, as is also shown in the slice 900 shown in Fig. 9 for the variable avr (INCLUSIVE-OR function 1002):

- an input variable n is incorrect after the first instruction 1 (element 1003),
- the eleventh instruction 11 is incorrect (element 1004),
- the value of the variable sum is incorrect before the eleventh instruction 11 (element 1005).

[0068] The variable sum is incorrect before the eleventh instruction 11 (element 1005) if at least one of the following events is satisfied (INCLUSIVE-OR function 1006):

- the variable sum is incorrect after the fourth instruction 4 (element 1007),
- ANDing 1008 the event that the loop body has been executed at least twice (element 1009) and the event that the tenth instruction 10 is incorrect (element 1010),
- ANDing 1011 the event that the loop body has been executed at least once (element 1012) and the result of INCLUSIVE-ORing 1013 the following four events:

    a) the ninth instruction 9 is incorrect (element 1014),

    b) the variable sum is incorrect after the fourth instruction 4 (element 1015),

    c) the variable i is incorrect after the fifth instruction 5 (element 1016),

d) the variable a is incorrect after the second instruction 2 (element 1017),

- INCLUSIVE-ORing 1018 the following events:

  e) the decision in accordance with the sixth instruction is incorrect (element 1019),

  f) the variable i is incorrect after the fifth instruction (element 1020),

  g) the variable n is incorrect after the first instruction (element 1021),

  h) ANDing 1022 the event that the 10th instruction is incorrect (element 1023) and the event that the loop body has been executed at least once (element 1024).

[0069] To provide for clearer illustration, the fault tree 1000 from Fig. 10 is altered such that events shown a plurality of times in the fault tree 1000 are combined to form one node of a cause-effect graph 1100 (cf. Fig. 11).

[0070] The fault tree 1000 shown in Fig. 10 is subjected to a fault tree analysis method, as described in DIN 25424-2, which analyzes an analysis of the computer program for a prescribed undesirable event.

[0071] The text below illustrates alternatives and further opportunities for application of the exemplary embodiment described above.

[0072] The overall fault tree produced using the method described above can be used for various purposes:

- description of the fault generation or propagation of incorrect action by a section of a computer program within the context of safety analysis or reliability analysis for the computer program,
- analysis of software fault mechanisms, for example within the context of test case generation.

[0073] The invention has been described in detail with particular reference to preferred embodiments thereof and examples, but it will be understood that variations and modifications can be effected within the spirit and scope of the invention.

SUBSTITUTE ABSTRACT

SYSTEM FOR DETERMINING A TOTAL ERROR DESCRIPTION OF AT LEAST ONE PART OF A COMPUTER PROGRAM

A section of a computer program is used to ascertain a control flow description and a data flow description, and program elements are selected from the section of the computer program. For each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element. The element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

MARKED-UP SUBSTITUTE SPECIFICATION

[Description]

TITLE OF THE INVENTION

[METHOD AND ARRANGEMENT] SYSTEM FOR DETERMINING A TOTAL ERROR DESCRIPTION OF AT LEAST ONE PART OF A COMPUTER [PROGRAMME] PROGRAM [AND COMPUTER PROGRAMME PRODUCT AND COMPUTER-READABLE STORAGE MEDIUM]

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application is based on and hereby claims priority to German Patent Application No. 19925239.4 filed on June 2, 2001, the contents of which are hereby incorporated by reference.

REFERENCE TO COMPUTER PROGRAM LISTING, COMPACT DISC APPENDIX

[0002] A compact disc is included herewith and incorporated by reference herein having thereon a computer program listing appendix in the ASCII uncompressed text format with ASCII carriage return, ASCII line feed and all control codes defined in ASCII, having computer compatibility with IBM PC/XT/AT or compatibles, having operating system compatibility with MS-Windows and including file PROGRA~6.TXT (Program-Listing.txt in Windows) of 130,121 bytes, created on November 29, 2001.

BACKGROUND OF THE INVENTION

[0003] The invention relates to a method and [an arrangement] a system for ascertaining an overall fault description for at least one section of a computer program, and also to a computer product and a computer-readable storage medium.

[0004] Such a method and such [an arrangement] a system are known from [1] [.] [1] N. Leveson, "Safety Verification of ADA Programs Using Software Fault Trees", IEEE Software, July 1991, pages 48-59, which discloses the practice of using computers to ascertain an overall fault description in the form of an overall fault tree for a computer program. For the computer program, a control flow description is ascertained in the form of a control flow graph. For various program elements of the computer program, a stored fault description associated with a respective stored reference element is used to ascertain an element fault description. The fault description for a reference element describes possible faults in the respective reference

element. The element fault descriptions in the form of element fault trees are used to ascertain the overall fault description, taking into account the control flow graph for the computer program.

[0005] The method and the [arrangement from] [1] system taught by Leveson have the following drawbacks, in particular. The overall fault tree ascertained is incomplete in terms of the faults examined and the causes thereof, and is therefore unreliable. Hence, this practice is not appropriate for use within the context of generating fault trees for a computer program for safety-critical applications. The individual fault trees associated with the reference elements are also incomplete and hence unreliable.

[0006] [2] M. Weiser, "Program Slicing", in: IEEE Transaction on Software Engineering, Vol. 10, No. 4, July 1984, pp. 352-357 provides an overview of "slicing". Slicing is the analysis carried out when searching for causes of incorrect action in a computer program. This procedure involves checking whether the incorrect action has been caused by an instruction currently under consideration. If this is not the case, the instructions which deliver data for or control the execution of the instruction are checked. This method is continued until no further operations exist, that is to say it gets to input data for the computer program. In slicing, "slices" are ascertained. A slice shows which instructions are affected in what way by a value under consideration. Below, the term slicing is always understood to mean backwardly directed slicing.

[0007] [3] P. Liggesmeyer, Modultest und Modulverifikation – State of the Art, Mannheim, Vienna, Zurich: BI Wissenschaftsverlag, 1990 discloses the practice of ascertaining a control flow description and a data flow description for a computer program. In [3] Liggesmeyer, this representation is used as an initial basis for "data-flow-oriented testing" of the computer program. The instructions (nodes) of the control flow graph are assigned data flow attributes (data flow description) which describe the nature of the data access operations contained in the instructions of the computer program. A distinction is drawn between write access operations and read access operations. Write access operations are referred to as definitions (def). Read access operations are referred to as a reference. If a read access operation takes place in a decision, this access operation is referred to as a predicative reference (p-use, predicate use). A read access operation during calculation of a value is referred to as a computational reference (c-use, computational use).

[0008] [4] DIN 25424-1: Fehlerbaumanalysen; Methoden und Bildzeichen, September 1981, which has a title that can be translated "Fault Tree Analyses; Methods and Graphic Symbols", discloses principles relating to a fault tree. A fault tree is to be understood, as described in [4]

DIN 25424-1, to mean a structure which describes logical relationships between input variables for the fault tree which lead to a prescribed undesirable event.

[0009] In addition, [5] DIN 25424-2: Fehlerbaumanalyse; Handrechenverfahren zur Auswertung eines Fehlerbaums, Berlin, Beuth Verlag GmbH, April 1990 which has a title that can be translated "Fault Tree Analysis; Manual Computation Methods for Evaluating a Fault Tree", discloses various methods for fault tree analysis.

SUMMARY OF THE INVENTION

[0010] The invention is based on the problem of ascertaining an overall fault description which is more reliable than ascertaining an overall fault tree in the manner known on the basis of the method [from] [1] taught by Leveson. [The problem is solved by the method and by the arrangement having the features in accordance with the independent claims and also by the computer product and the computer-readable storage medium having the features in accordance with the independent claims.]

[0011] In a method for ascertaining an overall fault description for at least one section of a computer program, using a computer, at least the section of the computer program is stored. A control flow description and a data flow description are ascertained for the section of the computer program, and program elements are selected from the section of the computer program. For each selected program element, a stored fault description is used to ascertain an element fault description. The fault description is associated with a respective reference element. The element fault description describes possible faults in the respective program element. A fault description for a reference element describes possible faults in the respective reference element. The element fault descriptions are used to ascertain the overall fault description, which takes into account the control flow description and the data flow description.

[0012] [An arrangement] A system for ascertaining an overall fault description for at least one section of a computer program has a processor which is set up such that the following method steps can be carried out:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,

- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,
- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

[0013] A computer program product comprises a computer-readable storage medium on which a program is stored which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,
- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

[0014] A computer-readable storage medium stores a program which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,

- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,
- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

[0015] The invention now makes it possible to ascertain a reliable overall fault description, which takes into account the peculiarities of a computer program, for a computer program or a section thereof. Since the overall fault description ascertained is much more reliable than the overall fault description which can be ascertained on the basis of the method [from] [1] taught by Leveson, the invention is also suitable for safety-critical applications, i.e. in particular for ascertaining an overall fault description for a safety-critical computer program. [Preferred developments of the invention can be found in the dependent claims.]

[0016] The control flow description and/or the data flow description may be in the form of a control flow graph or of a data flow graph, respectively.

[0017] The fault description may be in the form of a stored fault tree, and the element fault description can be ascertained as an element fault tree. In this case, the overall fault description can be ascertained as an overall fault tree. This development permits standardized representation of a fault description, which makes it much simpler for a user of the fault description to analyze same.

[0018] In one development, the overall fault description can be used for fault analysis in the section of the computer program. This development has the advantage, in particular, that automated, reliable fault analysis becomes possible, and if the fault descriptions are in the form of fault trees the fault description can even be analyzed in a manner "normalized" in accordance with the fault tree analysis methods.

[0019] In another refinement, the overall fault description is ascertained as an overall fault tree, and the overall fault tree is altered in terms of prescribable boundary conditions. The alteration can be made by adding a complementary fault tree.

BRIEF DESCRIPTION OF THE DRAWINGS

[0020] [An] These and other objects and advantages of the present invention will become more apparent and more readily appreciated from the following description of the exemplary embodiment of the invention [is] as explained in more detail below and [is] as illustrated in the figures., in which:

[0021] Figure 1 [shows] is a block diagram of a computer used to carry out the method in accordance with the exemplary embodiment;

[0022] Figure 2 [shows] is a flowchart showing the individual method steps of the method in accordance with the exemplary embodiment;

[0023] Figure 3 [shows an illustration of] is a general fault tree, as formed basically for a reference element;

[0024] Figures 4a to 4c [show] are flow diagrams for a control flow graph ([figure] Fig. 4a), a slice ([figure] Fig. 4b) and a fault tree ([figure] Fig. 4c) for an instruction sequence as a reference element of a computer program;

[0025] Figures 5a to 5c [show] are flow diagrams for a control flow graph ([figure] Fig. 5a), a slice ([figure] Fig. 5b) and a fault tree ([figure] Fig. 5c) for a selection sequence as a reference element of a computer program;

[0026] Figures 6a to 6c [show] are flow diagrams for a control flow graph ([figure] Fig. 6a), a slice ([figure] Fig. 6b) and a fault tree ([figure] Fig. 6c) for a loop as a reference element of a computer program;

[0027] Figure 7 [shows] is a control flow graph with a data flow graph for a computer program in accordance with the exemplary embodiment;

[0028] Figures 8a and 8b [show] are flow diagrams for a slice for the output of the variable max ([figure] Fig. 8a) and a slice for the variable avr ([figure] Fig. 8b) for the program in accordance with the exemplary embodiment;

[0029] Figure 9 [shows] is a flow diagram for the slice for the variable avr, in which a structure of the loop from the program in accordance with the exemplary embodiment is highlighted;

[0030] Figure 10 [shows] is a flow diagram for a fault tree for the assumption that the variable avr is incorrect;

[0031] Figure 11 [shows] is a flow diagram for the overall fault tree based on [figure] Fig. 10, where redundant events from the overall fault tree based on [figure] Fig. 10 have been combined into one event.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

[0032] Reference will now be made in detail to the preferred embodiments of the present invention, examples of which are illustrated in the accompanying drawings, wherein like reference numerals refer to like elements throughout.

[0033] Figure 1 shows a computer 100 used to carry out the method described below. The computer 100 has a processor 101 which is connected to a memory 102 via a bus 103. The bus 103 also has an input/output interface 106 connected to it.

[0034] The memory 102 stores a computer program 104 for which an overall fault description is ascertained in the manner described below. In addition, the memory 102 stores a program 105 which implements the method described below. The memory also stores fault descriptions 115 for different reference elements of a computer program. A fault description for a reference element describes possible faults in the respective reference element. Various reference elements and fault descriptions associated with the reference elements are explained in detail below.

[0035] The input/output interface 106 has a keyboard 108 connected to it via a first connection 107. A second connection 109 is used to connect the input/output interface 106 to a computer mouse 110, and a third connection 111 is used to connect the input/output interface 106 to a screen 112 on which the overall fault description ascertained for the computer program 104 is displayed. A fourth connection 113 is used to connect the input/output interface 106 to an external storage medium 114.

[0036] Figure 2 shows a block diagram illustrating the procedure in accordance with the exemplary embodiment described below.

[0037] The stored computer program 104 is used to ascertain a control flow graph 201 and a data flow graph 202 for the computer program 104.

[0038] Individual program elements are selected from the computer program (step 203). For each program element selected, a stored fault description associated with a reference element corresponding to the selected program element is used to ascertain an element fault description

(step 204). The element fault description describes possible faults in the respective selected program element.

[0039] On the basis of a fault event in the computer program (undesirable event), which fault event is prescribed by a user and needs to be examined, in a final step (step 205) an overall fault description for the computer program is ascertained, for the fault instance to be examined, from the element fault descriptions, taking into account the control flow graph and the data flow graph. The overall fault tree ascertained is displayed to the user on the screen 112.

[0040] Figure 3 shows the basic procedure for creating a fault tree, as used in the initial example in order to form the fault trees described below for the reference elements.

[0041] For an event 301 selected by a user, it is necessary to ascertain how the selected incorrect event can arise. In a computer program, incorrect output of a variable, as a selected incorrect event (undesirable event) 301, can be caused by a control flow fault 303 and/or a data fault 304 (INCLUSIVE-OR function 302).

[0042] A control flow fault 303 is to be understood to mean incorrect control of the processing of the respective variable.

[0043] The data flow fault 304 is to be understood to mean a fault which arises during processing as a result of incorrect data. The data flow fault 304 may originate in the processing step currently under consideration (block 306) and/or it may already have been present and may be maintained only by fault propagation (block 307) (INCLUSIVE-OR function 305).

[0044] On the basis of these considerations, the appropriate fault tree, a slice describing the instruction and a control flow graph are respectively illustrated below for the following elements of a computer program:

- an instruction sequence,
- a selection element,
- a loop element.

Instruction sequence

[0045] The instruction sequence 401 comprises the three instructions shown in [figure] Fig. 4a. In a first instruction 402, a first variable j is assigned the value 3 (j: = 3). A second instruction 403 assigns a second variable k the value 2 (k: = 2). A third instruction 404 forms a sum using the first variable and the second variable (i: = j + k).

[0046] In accordance with the practice disclosed in [2] Weiser, a slice 410 is formed for this instruction sequence 401, as shown in [figure] Fig. 4b. The first instruction 402 and the second instruction 403 both affect the third instruction 404, which is illustrated by two arrows 411, 412 in the slice 410.

[0047] For the control flow graph 401, the fault tree 420 shown in [figure] Fig. 4c is obtained for the following prescribed undesirable event 421: "Variable i is incorrect after the third instruction".

[0048] The incorrect event 421 may have been produced by a fault in the third instruction 404 under consideration itself, if the data up to this instruction step were correct (element 422 in [figure] Fig. 4c). The incorrect event 421 may also be caused by corrupt input data for the third instruction, however, i.e. as a result of INCLUSIVE-ORing 424 the events that the second variable k was incorrect after the second instruction (element 425) and/or that the first variable j was incorrect after the first instruction 402 (element 426). The result of the first INCLUSIVE-ORed function 424 is INCLUSIVE-ORed with the event that the third instruction is incorrect (INCLUSIVE-OR function 423).

Selection element

[0049] With a selection element as reference element, it is necessary to consider possibilities of fault in the data flows and in the control flows within the computer program.

[0050] [Figure] Figures 5a to [figure] 5c show a control flow graph 501 (cf. [figure] Fig. 5a), a slice 520 (cf. [figure] Fig. 5b) and a fault tree 540 (cf. [figure] Fig. 5c) for an If-Then-Else instruction as a selection element. The control flow graph 501 comprises the following six instructions:

- a first instruction 502, which assigns a first variable j the value 3 (j: = 3),
- a second instruction 503, which assigns a second variable k a prescribable value (k: = ...),
- a third instruction 504, which checks whether the second variable k has a value greater than 0; if the value of the second variable is greater than 0, the instruction branches to a fourth instruction 505, otherwise it branches to a fifth instruction 506,
- the fourth instruction 505, which assigns a third variable i the value of the second variable k (i: = k),
- a fifth instruction 506, which assigns the third variable i the value of the second variable k with a negative arithmetic sign (i: = -k),

- a sixth instruction 507, which processes the third variable i further in an arbitrary manner.

[0051] For the control flow graph 501 shown in [figure] Fig. 5a, the slice 520 shown in [figure] Fig. 5b is obtained for the selection element. Solid edges in the slice 520 show a data dependency between the different instructions. Dashed edges indicate control dependencies between the appropriate instructions.

[0052] The following definitions apply for the two edge types:

- dashed edges, referred to as control edges below, are directed from instructions which contain a predicative reference (failure constructs, loop control) to the directly controlled instructions, i.e. to those instructions which are executed only if the predicate has a particular value. Control edges are drawn only between the controlling instruction and directly interleaved instructions. If a controlled block contains a further interleaved control level, no control edges crossing more than one level are drawn. Since a control relationship is transitive, this indirect control can be inferred from the slice by utilizing the transitivity.
- solid edges, referred to as data flow edges below, are directed from instructions in which a variable is defined to instructions in which this variable is referenced. The variable under consideration cannot be defined again between the definition and the reference. This is referred to as a definition-free path for the variable under consideration.

[0053] The slice is ascertained by searching the control flow graph, counter to the edge direction, for a definition of the variable under consideration starting from the instruction containing the variable under consideration, for which the undesirable event is prescribed. If computational references exist for the definition, the method is continued recursively until no further additional nodes are found. The dependencies found in this way between instructions are data dependencies. If a node under consideration is contained in a block whose execution is controlled directly by a decision, this represents a control dependency. For the predicative references of the variables involved in the decision, nodes with appropriate definitions - that is to say data flow dependencies - are recursively sought which have other control dependencies.

[0054] Figure 5b shows the failure element's associated slice 520 with corresponding control edges and data flow edges.

[0055] Figure 5c shows the fault tree 540 for the prescribed event "the third variable i is incorrect before the 6th instruction" 541.

[0056] The following events result in the incorrect event 541 when INCLUSIVE-ORed 542:

- ANDing 543 the events that the decision in accordance with the third instruction 504 is true (element 544) and a result of INCLUSIVE-ORing 545 the events that the fourth instruction 505 is incorrect (element 546) and/or the first variable j is incorrect after the first instruction 502 (element 547);
- ANDing 550 the events that the decision in accordance with the third instruction 504 is false (element 551) and a result of INCLUSIVE-ORing 552 the events that the fifth instruction is incorrect (element 553) and/or that the first variable j is incorrect after the first instruction 502 (element 554);
- INCLUSIVE-ORing 560 the following events: the decision in accordance with the third instruction 504 is incorrect (element 561) and/or the second variable k is incorrect after the second instruction 503 (element 562).

Multiple selection element

[0057] A multiple selection element as reference element can be handled in accordance with the scheme described above by breaking down the multiple selection into a cascade of two-way selection elements processed in accordance with the procedure above, in order thus to ascertain a fault tree for a multiple selection element.

Loop

[0058] [Figure] Figures 6a to [figure] _6c show a fault tree 601 (cf. [figure] Fig. 6a), the corresponding slice 620 (cf. [figure] Fig. 6b) and the associated fault tree 640 (cf. [figure] Fig. 6c) for the reference element of a loop. The control flow graph 601 for a loop element comprises the following seven instructions:

- a first instruction 602, which assigns a first variable i the value 0 (i: = 0),
- a second instruction 603, which assigns a second variable j an unspecified value (j: = ...),
- a third instruction 604, which prescribes a further unspecified value for a third variable k (k: = ...),
- a fourth instruction 605, which, as a loop instruction, specifies a condition that a fifth instruction and a sixth instruction are executed until the value of the second variable is j > 0 (WHILE j > 0 DO),
- a fifth instruction 606, which assigns the first variable i a value which is obtained from the sum of the previous value of the first variable and the product of the second variable and the third variable (i: = i + k * j),

- a sixth instruction 607, which assigns the second variable j a value which is obtained by decreasing the original value of the second variable j by the value 1 (j: = j - 1),
- a seventh instruction 608, which processes the first variable i further in a prescribable manner (...: = i ...).

[0059] Figure 6b shows the corresponding slice 620 for the control flow graph 601 shown in [figure] Fig. 6a with associated control flow edges and data flow edges. The fault tree 640 shown in [figure] Fig. 6c is formed for the prescribed event 641 that the "first variable i is incorrect before the seventh instruction".

[0060] The fault tree 640 is obtained by INCLUSIVE-ORing 642 the following four events:

- a first event 643, which describes a situation in which the first variable i is incorrect after the first instruction 602,
- ANDing 644 the events that the loop body has been passed through at least twice (element 645) and the event that the sixth instruction 607 is incorrect (646),
- ANDing 650 the event that the loop body has been executed at least once (element 651) and INCLUSIVE-ORing 652 of the following four events:
    a) the fifth instruction 606 is incorrect (element 653),
    b) the first variable i is incorrect after the first instruction (element 654),
    c) the second variable j is incorrect after the second instruction (element 655),
    d) the third variable k is incorrect after the third instruction (element 656),
- INCLUSIVE-ORing 660 the following three events:
    e) the decision in accordance with the fourth instruction 605 is incorrect (element 661),
    f) the second variable j is incorrect after the second instruction 603 (element 662),
    g) ANDing 663 the events that the sixth instruction is incorrect (element 664) and the event that the loop body has been passed through at least once (element 665).

[0061] The fault trees described above, which are associated with the individual reference elements, are stored in the memory 102 as fault trees 115.

[0062] Figure 7 shows a control flow graph 700 for the following computer program:

```
input (n);
input (a);
```

```
max:=0;
sum:=0;
i:=2;
WHILE i =n DO
        IF max < a[i]
        THEN max:= a[i]
        sum:= sum + a[i]
        i:= i + 1
avr:= sum/n;
output (max);
output (avr);
```

[0063] For the control flow graph 700 comprising 13 instructions (reference symbols 1, 2, 3, ..., 13) which is shown in [figure] Fig. 7, [figure] Fig. 8a shows the associated slice 800 for the variable max and [figure] Fig. 8b shows the associated slice 810 for the variable avr. The numbering of the individual instructions in the slices corresponds to the numbering of the individual instructions in the control flow graph 700 from [figure] Fig. 7.

[0064] Figure 9 shows the slice 900 for the variable avr, as shown in [figure] Fig. 8b. The structure of the loop element contained in the program shown above is highlighted in bold. This structure corresponds to the slice shown in [figure] Fig. 6b for a loop element.

[0065] An overall fault tree 1000 for the computer program shown above is shown in [figure] Fig. 10. The overall fault tree for the computer program is produced by instantiating the appropriate fault tree associated with the reference element which corresponds to the selected program element.

[0066] By starting from the prescribed undesirable event and working backward, the overall fault tree 1000 is thus ascertained using the fault trees associated with the reference elements.

[0067] Figure 10 contains the fault tree 1000 relating to the event that "the variable avr is incorrect before the thirteenth instruction" (element 1001). The variable avr may be incorrect before the thirteenth instruction 13 on account of at least one of the following three events, as is also shown in the slice 900 shown in [figure] Fig. 9 for the variable avr (INCLUSIVE-OR function 1002):

- an input variable n is incorrect after the first instruction 1 (element 1003),
- the eleventh instruction 11 is incorrect (element 1004),

- the value of the variable sum is incorrect before the eleventh instruction 11 (element 1005).

[0068] The variable sum is incorrect before the eleventh instruction 11 (element 1005) if at least one of the following events is satisfied (INCLUSIVE-OR function 1006):

- the variable sum is incorrect after the fourth instruction 4 (element 1007),
- ANDing 1008 the event that the loop body has been executed at least twice (element 1009) and the event that the tenth instruction 10 is incorrect (element 1010),
- ANDing 1011 the event that the loop body has been executed at least once (element 1012) and the result of INCLUSIVE-ORing 1013 the following four events:
    a) the ninth instruction 9 is incorrect (element 1014),
    b) the variable sum is incorrect after the fourth instruction 4 (element 1015),
    c) the variable i is incorrect after the fifth instruction 5 (element 1016),
    d) the variable a is incorrect after the second instruction 2 (element 1017),
- INCLUSIVE-ORing 1018 the following events:
    e) the decision in accordance with the sixth instruction is incorrect (element 1019),
    f) the variable i is incorrect after the fifth instruction (element 1020),
    g) the variable n is incorrect after the first instruction (element 1021),
    h) ANDing 1022 the event that the 10th instruction is incorrect (element 1023) and the event that the loop body has been executed at least once (element 1024).

[0069] To provide for clearer illustration, the fault tree 1000 from [figure] Fig. 10 is altered such that events shown a plurality of times in the fault tree 1000 are combined to form one node of a cause-effect graph 1100 (cf. [figure] Fig. 11).

[0070] The fault tree 1000 shown in [figure] Fig. 10 is subjected to a fault tree analysis method, as described in [5] DIN 25424-2, which analyzes an analysis of the computer program for a prescribed undesirable event.

[0071] The text below illustrates alternatives and further opportunities for application of the exemplary embodiment described above.

[0072] The overall fault tree produced using the method described above can be used for various purposes:

- description of the fault generation or propagation of incorrect action by a section of a computer program within the context of safety analysis or reliability analysis for the computer program,
- analysis of software fault mechanisms, for example within the context of test case generation.

[0073] The invention has been described in detail with particular reference to preferred embodiments thereof and examples, but it will be understood that variations and modifications can be effected within the spirit and scope of the invention.

[The following publications are cited in this document:]

[1] [N. Leveson, Safety Verification of ADA Programs using software fault trees, IEEE Software, pages 48 to 59, July 1991]

[2] [Weiser M., *Program Slicing*, in: IEEE Transaction on Software Engineering, Vol. 10, No. 4, July 1984, pp. 352-357]

[3] [Liggesmeyer P., Modultest und Modulverifikation] [Module Test and Module Verification] [– State of the Art, Mannheim, Vienna, Zurich: BI Wissenschaftsverlag 1990]

[4] [DIN 25424-1: Fehlerbaumanalysen; Methoden und Bildzeichen] [Fault Tree Analyses; Methods and Graphic Symbols][, September 1981]

[5] [DIN 25424-2: Fehlerbaumanalyse; Handrechenverfahren zur Auswertung eines Fehlerbaums] [Fault Tree Analysis; Manual Computation Methods for Evaluating a Fault Tree] [, Berlin, Beuth Verlag GmbH, April 1990]

12/pats

Description

Method and arrangement for ascertaining an overall fault description for at least one section of a computer program, and computer program product and computer-readable storage medium

The invention relates to a method and an arrangement for ascertaining an overall fault description for at least one section of a computer program, and also to a computer product and a computer-readable storage medium.

Such a method and such an arrangement are known from [1].

[1] discloses the practice of using computers to ascertain an overall fault description in the form of an overall fault tree for a computer program. For the computer program, a control flow description is ascertained in the form of a control flow graph. For various program elements of the computer program, a stored fault description associated with a respective stored reference element is used to ascertain an element fault description. The fault description for a reference element describes possible faults in the respective reference element. The element fault descriptions in the form of element fault trees are used to ascertain the overall fault description, taking into account the control flow graph for the computer program.

The method and the arrangement from [1] have the following drawbacks, in particular. The overall fault tree ascertained is incomplete in terms of the faults examined and the causes thereof, and is therefore unreliable. Hence, this practice is not appropriate for use within the context of generating fault trees for a

computer program for safety-critical applications. The individual fault trees associated with

the reference elements are also incomplete and hence unreliable.

[2] provides an overview of "slicing". Slicing is the analysis carried out when searching for causes of incorrect action in a computer program. This procedure involves checking whether the incorrect action has been caused by an instruction currently under consideration. If this is not the case, the instructions which deliver data for or control the execution of the instruction are checked. This method is continued until no further operations exist, that is to say it gets to input data for the computer program. In slicing, "slices" are ascertained. A slice shows which instructions are affected in what way by a value under consideration. Below, the term slicing is always understood to mean backwardly directed slicing.

[3] discloses the practice of ascertaining a control flow description and a data flow description for a computer program. In [3], this representation is used as an initial basis for "data-flow-oriented testing" of the computer program. The instructions (nodes) of the control flow graph are assigned data flow attributes (data flow description) which describe the nature of the data access operations contained in the instructions of the computer program. A distinction is drawn between write access operations and read access operations. Write access operations are referred to as definitions (def). Read access operations are referred to as a reference. If a read access operation takes place in a decision, this access operation is referred to as a predicative reference (p-use, predicate use). A read access operation during calculation of a value is referred to as a computational reference (c-use, computational use).

[4] discloses principles relating to a fault tree. A

fault tree is to be understood, as described in [4], to mean a structure

which describes logical relationships between input variables for the fault tree which lead to a prescribed undesirable event.

In addition, [5] discloses various methods for fault tree analysis.

The invention is based on the problem of ascertaining an overall fault description which is more reliable than ascertaining an overall fault tree in the manner known on the basis of the method from [1].

The problem is solved by the method and by the arrangement having the features in accordance with the independent claims and also by the computer product and the computer-readable storage medium having the features in accordance with the independent claims.

In a method for ascertaining an overall fault description for at least one section of a computer program, using a computer, at least the section of the computer program is stored. A control flow description and a data flow description are ascertained for the section of the computer program, and program elements are selected from the section of the computer program. For each selected program element, a stored fault description is used to ascertain an element fault description. The fault description is associated with a respective reference element. The element fault description describes possible faults in the respective program element. A fault description for a reference element describes possible faults in the respective reference element. The element fault descriptions are used to ascertain the overall fault description, which takes into account the control flow description and the data flow description.

An arrangement for ascertaining an overall fault description for at least one section of a computer program has a processor which is set up such that the following method steps can be carried out:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,
- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

A computer program product comprises a computer-readable storage medium on which a program is stored which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault

description associated with a respective reference element is used to ascertain an element fault

description which describes possible faults in the respective program element,

- a fault description for a reference element describes possible faults in the respective reference element,
- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

A computer-readable storage medium stores a program which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description and a data flow description are ascertained for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element,
- the element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

The invention now makes it possible to ascertain a reliable overall fault description, which takes into account the peculiarities of a computer program, for a computer

program or a section thereof. Since the overall fault description ascertained is much more reliable than the overall

fault description which can be ascertained on the basis of the method from [1], the invention is also suitable for safety-critical applications, i.e. in particular for ascertaining an overall fault description for a safety-critical computer program.

Preferred developments of the invention can be found in the dependent claims.

The control flow description and/or the data flow description may be in the form of a control flow graph or of a data flow graph, respectively.

The fault description may be in the form of a stored fault tree, and the element fault description can be ascertained as an element fault tree. In this case, the overall fault description can be ascertained as an overall fault tree.

This development permits standardized representation of a fault description, which makes it much simpler for a user of the fault description to analyze same.

In one development, the overall fault description can be used for fault analysis in the section of the computer program.

This development has the advantage, in particular, that automated, reliable fault analysis becomes possible, and if the fault descriptions are in the form of fault trees the fault description can even be analyzed in a manner "normalized" in accordance with the fault tree analysis methods.

In another refinement, the overall fault description is ascertained as an overall fault tree, and the overall fault tree is altered in terms of prescribable boundary conditions.

The alteration can be made by adding a complementary fault tree.

An exemplary embodiment of the invention is explained in more detail below and is illustrated in the figures, in which:

Figure 1 shows a computer used to carry out the method in accordance with the exemplary embodiment;

Figure 2 shows a flowchart showing the individual method steps of the method in accordance with the exemplary embodiment;

Figure 3 shows an illustration of a general fault tree, as formed basically for a reference element;

Figures 4a to 4c show a control flow graph (figure 4a), a slice (figure 4b) and a fault tree (figure 4c) for an instruction sequence as a reference element of a computer program;

Figures 5a to 5c show a control flow graph (figure 5a), a slice (figure 5b) and a fault tree (figure 5c) for a selection sequence as a reference element of a computer program;

Figures 6a to 6c show a control flow graph (figure 6a), a slice (figure 6b) and a fault tree (figure 6c) for a loop as a reference element of a computer program;

Figure 7 shows a control flow graph with a data flow graph for a computer program in accordance with the exemplary embodiment;

Figures 8a and 8b show a slice for the output of the variable max (figure 8a) and a slice for the variable avr (figure 8b) for the program in accordance with the exemplary embodiment;

Figure 9 shows the slice for the variable avr, in which a structure of the loop from the program in accordance with the exemplary embodiment is highlighted;

Figure 10 shows a fault tree for the assumption that the variable avr is incorrect;

Figure 11 shows the overall fault tree based on figure 10, where redundant events from the overall fault tree based on figure 10 have been combined into one event.

Figure 1 shows a computer 100 used to carry out the method described below.

The computer 100 has a processor 101 which is connected to a memory 102 via a bus 103. The bus 103 also has an input/output interface 106 connected to it.

The memory 102 stores a computer program 104 for which an overall fault description is ascertained in the manner described below. In addition, the memory 102 stores a program 105 which implements the method described below. The memory also stores fault descriptions 115 for different reference elements of a computer program. A fault description for a reference element describes possible faults in the respective reference element. Various reference elements and fault descriptions associated with the reference elements are explained in detail below.

The input/output interface 106 has a keyboard 108 connected to it via a first connection 107. A second connection 109 is used to connect the input/output interface 106 to a computer mouse 110, and a third connection 111 is used to connect the input/output interface 106 to a screen 112 on which the overall fault description ascertained for the computer program 104 is displayed. A fourth connection 113 is used to connect the input/output interface 106 to an external storage medium 114.

Figure 2 shows a block diagram illustrating the procedure in accordance with the exemplary embodiment described below.

The stored computer program 104 is used to ascertain a control flow graph 201 and a data flow graph 202 for the computer program 104.

Individual program elements are selected from the computer program (step 203). For each program element selected, a stored fault description associated with a reference element corresponding to the selected program element is used to ascertain an element fault description (step 204). The element fault description describes possible faults in the respective selected program element.

On the basis of a fault event in the computer program (undesirable event), which fault event is prescribed by a user and needs to be examined, in a final step (step 205) an overall fault description for the computer program is ascertained, for the fault instance to be examined, from the element fault descriptions, taking into account the control flow graph and the data flow graph.

The overall fault tree ascertained is displayed to the user on the screen 112.

Figure 3 shows the basic procedure for creating a fault tree, as used in the initial example in order to form the fault trees described below for the reference elements.

For an event 301 selected by a user, it is necessary to ascertain how the selected incorrect event can arise. In a computer program, incorrect output of a variable, as a selected incorrect event

(undesirable event) 301, can be caused by a control flow fault 303 and/or a data fault 304 (INCLUSIVE-OR function 302).

A control flow fault 303 is to be understood to mean incorrect control of the processing of the respective variable.

The data flow fault 304 is to be understood to mean a fault which arises during processing as a result of incorrect data.

The data flow fault 304 may originate in the processing step currently under consideration (block 306) and/or it may already have been present and may be maintained only by fault propagation (block 307) (INCLUSIVE-OR function 305).

On the basis of these considerations, the appropriate fault tree, a slice describing the instruction and a control flow graph are respectively illustrated below for the following elements of a computer program:

- an instruction sequence,
- a selection element,
- a loop element.

Instruction sequence

The instruction sequence 401 comprises the three instructions shown in figure 4a. In a first instruction 402, a first variable j is assigned the value 3 (j: = 3). A second instruction 403 assigns a second variable k the value 2 (k: = 2). A third instruction 404 forms a sum using the first variable and the second variable (i: = j + k).

In accordance with the practice disclosed in [2], a slice 410 is formed for this instruction sequence 401, as shown in figure 4b. The first instruction 402 and the second instruction 403 both affect the third instruction 404, which is illustrated by two arrows 411, 412 in the slice 410.

For the control flow graph 401, the fault tree 420 shown in figure 4c is obtained for the following prescribed undesirable event 421:

"Variable i is incorrect after the third instruction".

The incorrect event 421 may have been produced by a fault in the third instruction 404 under consideration itself, if the data up to this instruction step were correct (element 422 in figure 4c). The incorrect event 421 may also be caused by corrupt input data for the third instruction, however, i.e. as a result of INCLUSIVE-ORing 424 the events that the second variable k was incorrect after the second instruction (element 425) and/or that the first variable j was incorrect after the first instruction 402 (element 426). The result of the first INCLUSIVE-ORed function 424 is INCLUSIVE-ORed with the event that the third instruction is incorrect (INCLUSIVE-OR function 423).

Selection element

With a selection element as reference element, it is necessary to consider possibilities of fault in the data flows and in the control flows within the computer program.

Figure 5a to figure 5c show a control flow graph 501 (cf. figure 5a), a slice 520 (cf. figure 5b) and a fault tree 540

(cf. figure 5c) for an If-Then-Else instruction as a selection element.

The control flow graph 501 comprises the following six instructions:

- a first instruction 502, which assigns a first variable j the value 3 (j: = 3),
- a second instruction 503, which assigns a second variable k a prescribable value (k: = ...),
- a third instruction 504, which checks whether the second variable k has a value greater than 0; if the value of the second variable is greater than 0, the instruction branches to a fourth instruction 505, otherwise it branches to a fifth instruction 506,
- the fourth instruction 505, which assigns a third variable i the value of the second variable k (i: = k),
- a fifth instruction 506, which assigns the third variable i the value of the second variable k with a negative arithmetic sign (i: = -k),
- a sixth instruction 507, which processes the third variable i further in an arbitrary manner.

For the control flow graph 501 shown in figure 5a, the slice 520 shown in figure 5b is obtained for the selection element.

Solid edges in the slice 520 show a data dependency between the different instructions.

Dashed edges indicate control dependencies between the appropriate instructions.

The following definitions apply for the two edge types:

- dashed edges, referred to as control edges below, are directed from instructions which contain a predicative reference (failure constructs, loop

control) to the directly controlled instructions, i.e. to those instructions which are executed only if

the predicate has a particular value. Control edges are drawn only between the controlling instruction and directly interleaved instructions. If a controlled block contains a further interleaved control level, no control edges crossing more than one level are drawn. Since a control relationship is transitive, this indirect control can be inferred from the slice by utilizing the transitivity.

- solid edges, referred to as data flow edges below, are directed from instructions in which a variable is defined to instructions in which this variable is referenced. The variable under consideration cannot be defined again between the definition and the reference. This is referred to as a definition-free path for the variable under consideration.

The slice is ascertained by searching the control flow graph, counter to the edge direction, for a definition of the variable under consideration starting from the instruction containing the variable under consideration, for which the undesirable event is prescribed. If computational references exist for the definition, the method is continued recursively until no further additional nodes are found. The dependencies found in this way between instructions are data dependencies. If a node under consideration is contained in a block whose execution is controlled directly by a decision, this represents a control dependency. For the predicative references of the variables involved in the decision, nodes with appropriate definitions - that is to say data flow dependencies - are recursively sought which have other control dependencies.

Figure 5b shows the failure element's associated slice 520 with corresponding control edges and data flow edges.

Figure 5c shows the fault tree 540 for the prescribed event "the third variable i is incorrect before the 6th instruction" 541.

The following events result in the incorrect event 541 when INCLUSIVE-ORed 542:

- ANDing 543 the events that the decision in accordance with the third instruction 504 is true (element 544) and a result of INCLUSIVE-ORing 545 the events that the fourth instruction 505 is incorrect (element 546) and/or the first variable j is incorrect after the first instruction 502 (element 547);
- ANDing 550 the events that the decision in accordance with the third instruction 504 is false (element 551) and a result of INCLUSIVE-ORing 552 the events that the fifth instruction is incorrect (element 553) and/or that the first variable j is incorrect after the first instruction 502 (element 554);
- INCLUSIVE-ORing 560 the following events: the decision in accordance with the third instruction 504 is incorrect (element 561) and/or the second variable k is incorrect after the second instruction 503 (element 562).

Multiple selection element

A multiple selection element as reference element can be handled in accordance with the scheme described above by breaking down the multiple selection into a cascade of two-way selection elements processed in accordance with the procedure above, in order thus to ascertain a fault tree for a multiple selection element.

Loop

Figure 6a to figure 6c show a fault tree 601 (cf. figure 6a), the corresponding slice 620 (cf. figure 6b) and the associated fault tree 640 (cf. figure 6c) for the reference element of a loop.

The control flow graph 601 for a loop element comprises the following seven instructions:

- a first instruction 602, which assigns a first variable i the value 0 (i: = 0),
- a second instruction 603, which assigns a second variable j an unspecified value (j: = ...),
- a third instruction 604, which prescribes a further unspecified value for a third variable k (k: = ...),
- a fourth instruction 605, which, as a loop instruction, specifies a condition that a fifth instruction and a sixth instruction are executed until the value of the second variable is j > 0 (WHILE j > 0 DO),
- a fifth instruction 606, which assigns the first variable i a value which is obtained from the sum of the previous value of the first variable and the product of the second variable and the third variable (i: = i + k * j),
- a sixth instruction 607, which assigns the second variable j a value which is obtained by decreasing the original value of the second variable j by the value 1 (j: = j - 1),
- a seventh instruction 608, which processes the first variable i further in a prescribable manner (...: = i ...).

Figure 6b shows the corresponding slice 620 for the control flow graph 601 shown in figure 6a with associated control flow edges and data flow edges.

The fault tree 640 shown in figure 6c is formed for the prescribed event 641 that the "first variable i is incorrect before the seventh instruction".

The fault tree 640 is obtained by INCLUSIVE-ORing 642 the following four events:

- a first event 643, which describes a situation in which the first variable i is incorrect after the first instruction 602,
- ANDing 644 the events that the loop body has been passed through at least twice (element 645) and the event that the sixth instruction 607 is incorrect (646),
- ANDing 650 the event that the loop body has been executed at least once (element 651) and INCLUSIVE-ORing 652 of the following four events:
  - a) the fifth instruction 606 is incorrect (element 653),
  - b) the first variable i is incorrect after the first instruction (element 654),
  - c) the second variable j is incorrect after the second instruction (element 655),
  - d) the third variable k is incorrect after the third instruction (element 656),
- INCLUSIVE-ORing 660 the following three events:
  - e) the decision in accordance with the fourth instruction 605 is incorrect (element 661),
  - f) the second variable j is incorrect after the second instruction 603 (element 662),
  - g) ANDing 663 the events that the sixth instruction is incorrect (element 664) and the event that the loop body has been passed through at least once (element 665).

The fault trees described above, which are associated with the individual reference elements, are stored in the memory 102 as fault trees 115.

Figure 7 shows a control flow graph 700 for the following computer program:

```
input (n);
input (a);
max:=0;
sum:=0;
i:=2;
WHILE i =n DO
      IF max < a[i]
      THEN max:= a[i]
      sum:= sum + a[i]
      i:= i + 1
avr:= sum/n;
output (max);
output (avr);
```

For the control flow graph 700 comprising 13 instructions (reference symbols 1, 2, 3, ..., 13) which is shown in figure 7, figure 8a shows the associated slice 800 for the variable max and figure 8b shows the associated slice 810 for the variable avr. The numbering of the individual instructions in the slices corresponds to the numbering of the individual instructions in the control flow graph 700 from figure 7.

Figure 9 shows the slice 900 for the variable avr, as shown in figure 8b. The structure of the loop element contained in the program shown above is highlighted in bold. This structure corresponds to the slice shown in figure 6b for a loop element.

An overall fault tree 1000 for the computer program shown above is shown in figure 10. The overall fault tree for the computer program is produced by instantiating the appropriate fault tree associated with the reference element which corresponds to the selected program element.

By starting from the prescribed undesirable event and working backward, the overall fault tree 1000 is thus ascertained using the fault trees associated with the reference elements.

Figure 10 contains the fault tree 1000 relating to the event that "the variable avr is incorrect before the thirteenth instruction" (element 1001). The variable avr may be incorrect before the thirteenth instruction 13 on account of at least one of the following three events, as is also shown in the slice 900 shown in figure 9 for the variable avr (INCLUSIVE-OR function 1002):

- an input variable n is incorrect after the first instruction 1 (element 1003),
- the eleventh instruction 11 is incorrect (element 1004),
- the value of the variable sum is incorrect before the eleventh instruction 11 (element 1005).

The variable sum is incorrect before the eleventh instruction 11 (element 1005) if at least one of the following events is satisfied (INCLUSIVE-OR function 1006):

- the variable sum is incorrect after the fourth instruction 4 (element 1007),
- ANDing 1008 the event that the loop body has been executed at least twice (element 1009) and the event that the tenth instruction 10 is incorrect (element 1010),
- ANDing 1011 the event that the loop body has been executed at least once (element 1012) and the result of INCLUSIVE-ORing 1013 the following four events:
  a) the ninth instruction 9 is incorrect (element 1014),
  b) the variable sum is incorrect after the fourth instruction 4 (element 1015),

c) the variable i is incorrect after the fifth instruction 5 (element 1016),

d) the variable a is incorrect after the second instruction 2 (element 1017),

- INCLUSIVE-ORing 1018 the following events:
  - e) the decision in accordance with the sixth instruction is incorrect (element 1019),
  - f) the variable i is incorrect after the fifth instruction (element 1020),
  - g) the variable n is incorrect after the first instruction (element 1021),
  - h) ANDing 1022 the event that the 10th instruction is incorrect (element 1023) and the event that the loop body has been executed at least once (element 1024).

To provide for clearer illustration, the fault tree 1000 from figure 10 is altered such that events shown a plurality of times in the fault tree 1000 are combined to form one node of a cause-effect graph 1100 (cf. figure 11).

The fault tree 1000 shown in figure 10 is subjected to a fault tree analysis method, as described in [5], which analyzes an analysis of the computer program for a prescribed undesirable event.

The text below illustrates alternatives and further opportunities for application of the exemplary embodiment described above.

The overall fault tree produced using the method described above can be used for various purposes:

- description of the fault generation or propagation of incorrect action by a section of a computer program within the context of safety analysis or reliability analysis for the computer program,
- analysis of software fault mechanisms, for example within the context of test case generation.

The text below shows a computer program in the programming language C++ which is used to implement the method in accordance with the exemplary embodiment:

```
#include "AS_GraphKante.h"
#include <iostream>


        ostream& operator << ( ostream& os, const GraphKanteC & Kante){
                os << Kante.KantenTyp;
                return os;
        }
////////////////////////////////////////////////////////////////////////////////
// Class for producing edges in the slice graph.
// There are two types of edges:
//              1. Control flow edges KFK
//              2. Data flow edges DFK
// This class also satisfies the nice-demands for the STL.
//
////////////////////////////////////////////////////////////////////////////////
// 1997-09-12  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifndef _GraphNodeHeader
#define _GraphNodeHeader

#include <iostream>
#include "KFGListNode.h"

using namespace std;

class  GraphNodeC : public KFGListNodeC {
public:
        //GraphNodeC()                                          {};
        friend ostream& operator << ( ostream& os, const GraphNodeC& Node);

};

#endif


////////////////////////////////////////////////////////////////////////////////
// Class for producing a slice. This class inherits an empty object from the
// Graph class.
// The corners of the graph/slice have the same structure as the lists in the CFG.
// The edges are of type KantenTypT; a GraphKanteC class could not be incorporated
// into the Graph class, for reasons which were not apparent to me.
////////////////////////////////////////////////////////////////////////////////
// 1997-09-12  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _SliceHeader
#else
#define _SliceHeader

#include "graph.h"
#include "AS_GraphKante.h"
//#include "AS_GraphNode.h"
#include "KFGListNode.h"
#include "KFGList.h"

//using namespace std;
//typedef enum {DFK, KFK} KantenTypT;

```

```
class SliceC : public Graph<KFGListNodeC,GraphKanteC> {

public:

        SliceC() {};

        void sliceForLoops(KFGListC & L2, KFGListC::iterator LOOPIT);

        KFGListC::iterator defineVariableToSlice(KFGListC & L2);

        //void buildTestGraph(KFGListC & L2);

        void findAllDefs(KFGListC & L1, KFGListC::iterator ITER);

        KFGListC::iterator findUpperLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGP_UseListC::iterator PUSEIT);

        KFGListC::iterator findLowerLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGListC::iterator UPPERLIMIT);

        KFGListC::iterator findLowerLimitFromCUse(KFGListC & L1, KFGListC::iterator
ITER, KFGListC::iterator UPPERLIMIT);

        KFGListC::iterator findUpperLimitFromCUse(KFGListC & L1, KFGListC::iterator
ITER, KFGUseListC::iterator USEIT);

        int checkForNodes(int NodeNumber);

        void KFKPUseToCUse(KFGListC::iterator PUSEIT, int SchleifenEnde);

        void findDefToCUse(KFGListC & L1, KFGListC::iterator ITER);

        void defInLoop(KFGListC & L1, KFGListC::iterator ITER);

        void startBuildSlice(KFGListC & L1);

        void sliceAusgeben();

};

#endif


////////////////////////////////////////////////////////////////////////////
// Taken with a few alterations from "The C++ Standard Template Library".
//
// Template Class Graph for producing directional or nondirectional graphs.
// The graph comprises a vector E for all corners. Each vector element comprises,
// in turn, a pair: the corner and the set of successors.
// The set of successors is represented by the STL data type map; the key for an
// edge type/edge value is the number of a subsequent corner.
//
////////////////////////////////////////////////////////////////////////////
#ifndef _graphHeader
#define _graphHeader
#include <assert.h>
#include <map>
#include <stack>
#include <vector>
#include "checkvec.h"
#include <iostream>
#include "AS_GraphNode.h"
using namespace std;

// Empty parameter class with minimum set of operations,
// if no edge weights are required.
struct Empty
{
    public:
      Empty(int=0) {}
      bool operator<(const Empty&) const { return true;}
};
// Empty operations so that in/output can be formulated in general terms
//ostream& operator<<(ostream& os, const Empty&) { return os;}
```

```
//istream& operator>>(istream& is, Empty& ) { return is;}


template<class Eckentyp, class Kantentyp>
class Graph
{
    public:
      typedef map<int,Kantentyp, less<int> > Nachfolger;
      typedef pair<Eckentyp, Nachfolger> Ecke;
      typedef checkedVector<Ecke> Graphtyp;
         //typedef vector<Ecke> Graphtyp;
      typedef Graphtyp::iterator iterator;
      typedef Graphtyp::const_iterator const_iterator;

      Graph(bool g, ostream& os = cerr)
      : gerichtet(g), pOut(&os) {}

         Graph() {}

      size_t size()       const { return C.size(); }
      bool istGerichtet() const { return gerichtet;}
      iterator begin()     { return C.begin();}
      iterator end()       { return C.end();}
      Ecke& operator[](int i) { return C[i];}

      size_t AnzahlKanten();
      int  insert(const Eckentyp& e);
      void insert(const Eckentyp& e1, const Eckentyp& e2,
                  const Kantentyp& Wert);
      void verbindeEcken(int e1, int e2,  // using corner numbers
                  const Kantentyp& Wert);
         void setgerichtet (bool wert); // new method for
directional/nondirectional graphs,
                                                          // although it
is not used.
      void check(ostream& = cout);
      void ZyklusUndZusammenhang(ostream& = cout);

    private:
      bool gerichtet;
      Graphtyp C;          // Container
      ostream* pOut;
};

//Function which checks whether a directional or nondirectional graph is involved,
//and outputs the number of corners and edges.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::check(ostream& os)
{
    os << "The graph is ";
    //if(!istGerichtet())
    //    os << "non";
    os << "directional and has "
           << size() << " nodes and "
           << AnzahlKanten() << " edges\n";
    //ZyklusUndZusammenhang(os);
}

//Method which sets a value determining whether the graph is directional or
nondirectional.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::setgerichtet(bool wert)
{
       gerichtet = wert;
}

//Function calculates the number of edges in graph
template<class Eckentyp, class Kantentyp>
size_t Graph<Eckentyp,Kantentyp>::AnzahlKanten()
{
    size_t Kanten = 0;
    iterator temp = begin();
    while(temp != end())
         Kanten += (*temp++).second.size();
    //if(!gerichtet)
    //    Kanten /= 2;
```

```
    return Kanten;
}

//Insert a corner into the graph if it does not yet exist.
template<class Eckentyp, class Kantentyp>
int Graph<Eckentyp,Kantentyp>::insert(const Eckentyp& e)
{
    for(int i = 0; i < size(); ++i)
       if(e == C[i].first)
          return i;

    // if not found, insert:
    C.push_back(Ecke(e, Nachfolger()));
    return size()-1;
}


//Insert an edge into the graph by first inserting the corners.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::insert(const Eckentyp& e1,
                                       const Eckentyp& e2,
                                       const Kantentyp& Wert)
{
    int pos1 = insert(e1);
    int pos2 = insert(e2);
    verbindeEcken(pos1, pos2, Wert);
}


//Connect the two newly inserted corners by means of an edge.
template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp,Kantentyp>::verbindeEcken(
            int pos1, int pos2, const Kantentyp& Wert)
{
    (C[pos1].second)[pos2] = Wert;
    //if(!gerichtet)  // automatically enter opposite direction as well
    //  (C[pos2].second)[pos1] = Wert;
}

/* ZyklusUndZusammenhang() uses the depth search.
   In contrast to CLR S. 478, recursion has not been used, because it involved the
   occurrence of stack overflow with large graphs (e.g. MILES.DAT
   at more than 40 nodes). Simulating recursion using a separate stack makes it
   possible to process the whole MILES.DAT file (128 nodes). The stack depth
   corrresponds to the number of edges + 1 in the case of nondirectional
   graphs.
*/

template<class Eckentyp, class Kantentyp>
void Graph<Eckentyp, Kantentyp>::ZyklusUndZusammenhang(ostream& os)
{
   int Zyklen = 0;
   int Komponentenanzahl = 0;
   stack<int,vector<int> > EckenStack;  // corners to be visited

   // assign all corners the state nichtBesucht
   enum EckStatus {nichtBesucht, besucht, bearbeitet};
   vector<EckStatus> Eckenzustand(size(), nichtBesucht);

   // visit all corners
   for(int i = 0; i < size(); ++i)
      if(Eckenzustand[i] == nichtBesucht)
      {
         Komponentenanzahl++;
         // deposit on the stack for the purposes of processing
         EckenStack.push(i);

         // Process stack
         while(!EckenStack.empty())
         {
            int dieEcke = EckenStack.top();
            EckenStack.pop();
            if(Eckenzustand[dieEcke] == besucht)
               Eckenzustand[dieEcke] = bearbeitet;
            else
```

```
                if(Eckenzustand[dieEcke] == nichtBesucht)
                {
                   Eckenzustand[dieEcke] = besucht;
                   // make a note of new corner, for bearbeitet identifier
                   EckenStack.push(dieEcke);

                   // make a note of successor:
                   Graph<Eckentyp, Kantentyp>::Nachfolger::iterator
                      start  = operator[](dieEcke).second.begin(),
                      ende   = operator[](dieEcke).second.end();
                   while(start != ende)
                   {
                        int Nachf = (*start).first;
                        if(Eckenzustand[Nachf] == besucht)
                        {
                            ++Zyklen;   // somebody has already been here!
                            (*pOut) << "at least corner "
                            << operator[](Nachf).first
                            << " is in a cycle\n";
                        }
                        if(Eckenzustand[Nachf] == nichtBesucht)
                           EckenStack.push(Nachf);
                        ++start;
                   }
                }
            } // Stack Empty?
        }     // for() ... if(Eckenzustand...

    if(gerichtet)
    {  if(Komponentenanzahl == 1)
          os << "The graph is very cohesive.\n";
       else
          os << "The graph is not or not very "
                "cohesive.\n";
    }
    else
       os << "The graph has "
             << Komponentenanzahl
             << " component(s)." << endl;

    os << "The graph has ";
    if(Zyklen == 0)
      os << "no ";
    os << "cycles." << endl;
}

//Output of the graph.
template<class Eckentyp, class Kantentyp>
ostream& operator<<(ostream& os,
                    Graph<Eckentyp,Kantentyp>& G)
{
       ofstream Ziel("FaultTree.uwg");
       ostream_iterator<uwgknotenC> POSIT(Ziel);
       ostream_iterator<uwgknotenC> POSIT2(Ziel);
    // Display the corners with successors
       Ziel << "%%UWG" << endl << "\"V0.1\"~\"fault tree\"~\"A.Steinhorst\"" <<
endl << "%%BEGIN" << endl;
    for(int i = 0; i < G.size(); ++i)
    {
             POSIT++ = G[i].first;
             /*os << G[i].first << " <";
        Graph<Eckentyp,Kantentyp>::Nachfolger::iterator
             startN = G[i].second.begin(),
             endeN  = G[i].second.end();
        while(startN != endeN)
        {
            os << G[(*startN).first].first << ' '     // Ecke
               << (*startN).second << ' ';   // Kantenwert
                    *POSIT2++ = G[(*startN).first].first;
                    //*KANTEIT++ = (*startN).second;
            ++startN;
        }
        os << ">\n";*/
             *POSIT++;
    }
        for(int u = 0; u < G.size(); ++u) {
             Graph<Eckentyp,Kantentyp>::Nachfolger::iterator
```

```
                    startN = G[u].second.begin(),
                    endeN  = G[u].second.end();
                while (startN != endeN) {
                    Ziel << "%%EDGE:" << G[u].first.getGatterIdent() << ","
                    << G[(*startN).first].first.getGatterIdent() << "/0;" << endl;
                    ++startN;
                }
        }
        Ziel << "%%PROBSLIST" << endl << "%%END" << endl;
    return os;
}

#endif

#ifdef _KFGDefHeader

#else

#define _KFGDefHeader


#include <string>

#include <iostream>


using namespace std;


        typedef char StringT [256];


        class KFGDefC {

        private:


                StringT Def;

                int ScopeLevelD;

        public:
                KFGDefC()                                   { strcpy (Def, "-- unknown --");
                                                              ScopeLevelD = 0; };

                KFGDefC(char _Def [], int _ScopeLevelD)
                                                            { strcpy (Def, _Def);
                                                              ScopeLevelD =
_ScopeLevelD; };
                void setDef(char _Def [])     { strcpy (Def, _Def); };

                char* getDef()                              { return Def;  };

                int getScopeLevelD()          { return ScopeLevelD; };

                bool operator == (const KFGDefC& other) const
                                                            { return ((strcmp(Def,
other.Def) == 0)
                                                            & (ScopeLevelD
== other.ScopeLevelD)); };

                bool operator != (const KFGDefC& other) const
                                                            { return !(*this ==
other);};

                bool operator < (const KFGDefC& other) const
                                                            { return (strcmp(Def,
other.Def) < 0); };

                bool operator > (const KFGDefC& other) const
```

```
                                                                { return (strcmp(Def,
other.Def) > 0); };

                friend ostream& operator << ( ostream& os, const CFGDefC& Node);
        };

#endif

////////////////////////////////////////////////////////////////////////////////
// Class for auxiliary list in order to produce a CFG therefrom.
// This class satisfies the nice-demands for the STL;
// i.e. the following is true for a class T:
// 0. It supports the copy constructor T (const T&),
// 1. the assignment operator T& operator= (const T&),
// 2. the comparison operator bool operator== (const T, &const T&) and
// 3. the unequal-to operator!=
// in such a way that:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      In addition, all functions, in particular getName, are equality preserving, i.e.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGLineNodeHeader
#else
#define _KFGLineNodeHeader

#include <string>
#include <iostream>

using namespace std;

        typedef char StringL [1000];

        class KFGLineNodeC {
        private:

                StringL Name;
                int LineNumber;

        public:

                KFGLineNodeC()                          { strcpy (Name, "-- unknown --
");
                                                          LineNumber = 0;
};
                KFGLineNodeC(char _Name [])    { strcpy (Name, _Name);
                                                          LineNumber =
0;};            KFGLineNodeC(char _Name [], int _LineNumber)
                                                                { strcpy (Name,
_Name);
                                                                  LineNumber =
_LineNumber; };

                void setName(char _Name [])         { strcpy (Name, _Name);      };

                char* getName()                                 { return Name; };

                int getLineNumber()                             { return LineNumber; };

                bool operator == (const KFGLineNodeC& other) const
                                                                { return
((strcmp(Name, other.Name) == 0)
```

```
                                                        & (LineNumber ==
other.LineNumber)); };

                bool operator != (const KFGLineNodeC& other) const
                                                        { return !(*this
== other);      };

                bool operator < (const KFGLineNodeC& other) const
                                                                { return
(strcmp(Name, other.Name) < 0); };

                bool operator > (const KFGLineNodeC& other) const
                                                                { return
(strcmp(Name, other.Name) > 0); };

                friend ostream& operator << ( ostream& os, const KFGLineNodeC&
Node);
        };

#endif

////////////////////////////////////////////////////////////////////////////////
// Class for the list which shows the CFG.
// This class satisfies the nice-demands for the STL;
// i.e. the following is true for a class T:
// 0. It supports the copy constructor T (const T&),
// 1. the assignment operator T& operator= (const T&),
// 2. the comparison operator bool operator== (const T, &const T&) and
// 3. the unequal-to operator!=
// in such a way that:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      In addition, all functions, in particular getStatement, are equality
        preserving, i.e.
// (g) { a == b  == > a.getStatement() == b.getStatement() }
////////////////////////////////////////////////////////////////////////////////
// 1997-09-01  Andreas Steinhorst
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGListHeader
#else
#define _KFGListHeader

#include <string>
#include <iostream>
#include <stdio.h>
#include <list>
#include <fstream>
#include <iterator>
#include "KFGListNode.h"
#include "KFGTokenList.h"
#include "KFGProgList.h"

using namespace std;

class KFGListC :  public list<KFGListNodeC> {

        int Anzahl_Defs, Anzahl_Uses, Anzahl_P_Uses, Anzahl_Deklarationen;
        int Schleifenentscheidungen, Entscheidungen, Zaehlschleifenentscheidungen;
        int Anweisungen;

public:

        KFGListC() {};

        void TokenList2KFGList(KFGTokenListC & L1);

        void KnotenNummern();

        void KnotenIdentifizierer(KFGListC::iterator KFG);

        void ListeAusgeben();
```

```
        void addLineInToList();

        void addLineToKFG(KFGProgListC & LP);

        int zaehleDeklarationen(KFGTokenListC & TL);

        void basisgroessenInDatei();

};

class KFGDefListC : public list<KFGDefC> {};

class KFGUseListC : public list<KFGUseC> {};

class KFGP_UseListC : public list<KFGUseC> {};

#endif

#include <string>
#include <iostream>
#include <list>
#include "KFGNode.h"

using namespace std;

typedef list<KFGNodeC> KFGListeT;

extern void KFGListeAusgeben(KFGListeT);

////////////////////////////////////////////////////////////////////////////////
///
// Class for the nodes in the control flow graph.
// This class satisfies the nice-demands for the STL;
// i.e. the following is true for a class T:
// 0. It supports the copy constructor T (const T&),
// 1. the assignment operator T& operator= (const T&),
// 2. the comparison operator bool operator== (const T, &const T&) and
// 3. the unequal-to operator!=
// in such a way that:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      In addition, all functions, in particular getStatement, are equality
      preserving, i.e.
// (g) { a == b  == > a.getStatement() == b.getStatement() }
////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGNodeListHeader
#else
#define _KFGNodeListHeader

#include <string>
#include <iostream>
#include <list>
#include "KFGUse.h"
#include "KFGDef.h"


using namespace std;

        typedef char StringT [256];
        typedef char CodeLineT[256];
        typedef enum
{NO,LC,BL,EL,IFC,BTh,ETh,BEl,EEl,OP,DOWL,DOWLC,SWITCH,CASE,ENDCASE,BDEFA,ENDDEFA,RE
TURN,BREAK} KFGNodeTypeT;
        typedef enum {LOOP, THEN, ELSE, NONE} KnotenIdentT;

        class KFGListNodeC {
```

```
        protected:
        //private:

                static int DummyNodeNr;
                int NodeNr;
                int Level;
                int KnotenNr;
                int LineNr;
                StringT Statement;
                CodeLineT CodeLine;
                KFGNodeTypeT KFGNodeType;
                KnotenIdentT KnotenIdent ;

        public:

                list<KFGUseC> Uses;
                list<KFGDefC> Defs;
                list<KFGUseC> P_Uses;

                KFGListNodeC()                { strcpy (Statement, "-- unknown --");
                                                  Level = 0;
                                                  KFGNodeType =
    NO;
                                              NodeNr = DummyNodeNr++;

    KnotenNr = 0;
                                                  KnotenIdent =
    NONE;
                                                  LineNr = 0;
                                            strcpy (CodeLine, "--
    unknown--"); };

                KFGListNodeC(char _Statement [], KFGNodeTypeT _KFGNodeType)
                                                { strcpy (Statement,
    _Statement);
                                                     Level = 0;
                                                  KFGNodeType =
    _KFGNodeType;
                                              NodeNr = DummyNodeNr++;

    KnotenNr = 0;
                                                  KnotenIdent =
    NONE;
                                                  LineNr = 0;
                                                         strcpy
    (CodeLine, "--unknown--"); };

                KFGListNodeC(char _Statement [], KFGNodeTypeT _KFGNodeType, int
    _Level)
                                              { strcpy (Statement,
    _Statement);
                                                     Level = _Level;
                                                  KFGNodeType =
    _KFGNodeType;
                                              NodeNr = DummyNodeNr++;

    KnotenNr = 0;
                                                  KnotenIdent =
    NONE;
                                                  LineNr = 0;
                                            strcpy (CodeLine, "--
    unknown--"); };

                KFGListNodeC(char _Statement [], KFGNodeTypeT _KFGNodeType, int
    _Level, int _LineNr)
                                              { strcpy (Statement,
    _Statement);
                                                     Level = _Level;

    KFGNodeType = _KFGNodeType;
                                              NodeNr = DummyNodeNr++;
                                                  KnotenNr = 0;

    KnotenIdent = NONE;
                                                                 LineNr
    = _LineNr;
```

```
                                                                    strcpy
(CodeLine, "--unknown--"); };
                void setStatement(char _Statement [])
                                                                    { strcpy
(Statement, _Statement);      };
                void setCodeLine(char _CodeLine [])
                                                                    { strcpy
(CodeLine, _CodeLine); };
                void setKFGKnotenNummer(int _KnotenNr)
                                                                    {
KnotenNr = _KnotenNr; };
                void setKnotenIdent (KnotenIdentT _KnotenIdent)
                                                                    {
KnotenIdent = _KnotenIdent; };
                int getKnotenNummer()          { return KnotenNr; };
                char* getStatement()                { return Statement;    };
                char* getCodeLine()                         { return CodeLine; };
                int getLevel()                              { return Level; };
                int getNodeNr()                             { return NodeNr; };
                int getLineNr()                             { return LineNr; };
                KnotenIdentT getKnotenIdent()       { return KnotenIdent; };
                KFGNodeTypeT getKFGNodeType() { return KFGNodeType; };
                bool operator == (const KFGListNodeC& other) const
                                                                    { return
((strcmp(Statement, other.Statement) == 0)
                                                                      &
(Level == other.Level)
                                                                      &
(KFGNodeType == other.KFGNodeType)
                                                                      &
(NodeNr == other.NodeNr) ); };
                bool operator != (const KFGListNodeC& other) const
                                                                    { return !(*this ==
other); };
                bool operator < (const KFGListNodeC& other) const
                                                                    { return
(strcmp(Statement, other.Statement) < 0); };
                bool operator > (const KFGListNodeC& other) const
                                                                    { return
(strcmp(Statement, other.Statement) > 0); };
                friend ostream& operator << ( ostream& os, const KFGListNodeC&
Node);
        };
#endif
//#include <string>
//#include <iostream>
//#include "KFGList.h"

//using namespace std;

//class KFGListOpC : public KFGListeC {

//public:

//      void KFGListOut(KFGListeT & L)
//      {
//              KFGListeT::iterator I = L.begin();
//              while(I != L.end())
```

```
//                    cout << *I++ << ' ';
//              cout << " size() = " <<  L.size() << endl;
//      }
//};
5

////////////////////////////////////////////////////////////////////////////////
///
// Class for auxiliary list in order to produce a CFG therefrom.
10  // This class satisfies the nice-demands for the STL;
// i.e. the following is true for a class T:
// 0. It supports the copy constructor T (const T&),
// 1. the assignment operator T& operator= (const T&),
// 2. the comparison operator bool operator== (const T, &const T&) and
15  // 3. the unequal-to operator!=
// in such a way that:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
20  // (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      In addition, all functions, in particular getName, are equality preserving,
25      i.e.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
////////////////////////////////////////////////////////////////////////////////
30  // 1997-08-25  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGNodeHeader
#else
#define _KFGNodeHeader
35

#include <string>
#include <iostream>

using namespace std;
40

        typedef char StringT [256];

        class KFGNodeC {
        private:
45
                StringT Name;
                int ScopeLevel;


        public:
50
                KFGNodeC()                              { strcpy (Name, "-- unknown --
"); ScopeLevel = 0;
                                                          /* cout << "NodeC
produced" << endl ; */};
55              KFGNodeC(char _Name [] )                { strcpy (Name, _Name);
                                                          ScopeLevel = 0; };

                KFGNodeC(char _Name [], int _ScopeLevel)
                                                                { strcpy (Name,
60  _Name);
                                                                ScopeLevel =
_ScopeLevel;};
                void setName(char _Name [])   { strcpy (Name, _Name);       };
65
                char* getName()                         { return Name; };

                int getScopeLevel()                     { return ScopeLevel; };

70              bool operator == (const KFGNodeC& other) const
                                                                { return
((strcmp(Name, other.Name) == 0)
                                                                  &
(ScopeLevel == other.ScopeLevel)); };
75
                bool operator != (const KFGNodeC& other) const
```

```
                                                                    { return
!(*this == other);     };
5               bool operator < (const KFGNodeC& other) const
                                                        { return (strcmp(Name,
other.Name) < 0); };

                bool operator > (const KFGNodeC& other) const
10                                                      { return (strcmp(Name,
other.Name) > 0); };

                friend ostream& operator << ( ostream& os, const KFGNodeC& Node);
        };
15
#endif

#ifdef _KFGNodelHeader
#else
20 ////////////////////////////////////////////////////////////////////////////////
// Class for auxiliary list in order to produce a CFG therefrom.
// This class satisfies the nice-demands for the STL;
// i.e. the following is true for a class T:
// 0. It supports the copy constructor T (const T&),
25 // 1. the assignment operator T& operator= (const T&),
// 2. the comparison operator bool operator== (const T, &const T&) and
// 3. the unequal-to operator!=
// in such a way that:
// (a) { TRUE } T a(b) {a == b}
30 // (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
35 //
//      In addition, all functions, in particular getName, are equality preserving,
        i.e.
// (g) { a == b  == > a.getName() == b.getName() }
//
40 // (C) 1997 Siemens AG
//
////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
45 #ifdef _KFGNodelHeader
#else
#define _KFGNodelHeader

#include <string>
50 #include <iostream>

using namespace std;

        typedef char StringT [256];
55
        class KFGListNodeC {
        private:

                StringT Name;
60              int ScopeLevel;
                list <KFGUseC> Uses;
                list <KFGDefC> Defs;

        public:
65
                KFGListNodeC()                                  { strcpy (Name,
"-- unknown --"); ScopeLevel = 0;
                                                                  /* cout
<< "NodeC produced" << endl ; */};
70
                KFGListNodeC(char _Name [] )            { strcpy (Name, _Name);
                                                          ScopeLevel = 0; };

                KFGListNodeC(char _Name [], int _ScopeLevel)
75                                                      { strcpy (Name,
_Name);
```

```
                                                        ScopeLevel =
_ScopeLevel;};

                void setName(char _Name [])   { strcpy (Name, _Name);       };
5
                char* getName()                                 { return Name; };

                int getScopeLevel()                             { return ScopeLevel; };

10              bool operator == (const KFGListNodeC& other) const
                                                                { return ((strcmp(Name,
other.Name) == 0)
                                                                        & (ScopeLevel
== other.ScopeLevel)); };
15
                bool operator != (const KFGListNodeC& other) const
                                                                { return !(*this ==
other); };

20              bool operator < (const KFGListNodeC& other) const
                                                                { return (strcmp(Name,
other.Name) < 0); };

                bool operator > (const KFGListNodeC& other) const
25                                                              { return (strcmp(Name,
other.Name) > 0); };

                friend ostream& operator << ( ostream& os, const KFGListNodeC&
Node);
30      };

#endif

#ifdef _KFGNode1Header
35  #else
    ////////////////////////////////////////////////////////////////////////////
    // Class for auxiliary list in order to produce a CFG therefrom.
    // This class satisfies the nice-demands for the STL;
    // i.e. the following is true for a class T:
40  // 0. It supports the copy constructor T (const T&),
    // 1. the assignment operator T& operator= (const T&),
    // 2. the comparison operator bool operator== (const T, &const T&) and
    // 3. the unequal-to operator!=
    // in such a way that:
45  // (a) { TRUE } T a(b) {a == b}
    // (b) { TRUE } a = b  {a == b}
    // (c) { a == a}
    // (d) { a == b <==> b == a }
    // (e) { (a == b) AND ( b == c ) ==> (a == c) }
50  // (f) { a != b <==> NOT ( a== b) };
    //
    //      In addition, all functions, in particular getName, are equality preserving,
         i.e.
    // (g) { a == b  == > a.getName() == b.getName() }
55  //
    // (C) 1997 Siemens AG
    //
    ////////////////////////////////////////////////////////////////////////////
    // 1997-08-25  Andreas Steinhorst     -
60  ////////////////////////////////////////////////////////////////////////////
    #ifdef _KFGNode1Header
    #else
    #define _KFGNode1Header

65  #include <string>
    #include <iostream>

    using namespace std;

70          typedef char StringT [256];

            class KFGListNodeC {
            private:

75                  StringT Name;
```

```
                int ScopeLevel;
                list <KFGUseC> Uses;
                list <KFGDefC> Defs;

        public:

                KFGListNodeC()                                          { strcpy (Name,
"-- unknown --"); ScopeLevel = 0;
                                                                /* cout << "NodeC
produced" << endl ; */};

                KFGListNodeC(char _Name [] )            { strcpy (Name, _Name);
                                                          ScopeLevel = 0; };

                KFGListNodeC(char _Name [], int _ScopeLevel)
                                                                { strcpy (Name,
_Name);
                                                              ScopeLevel =
_ScopeLevel;};

                void setName(char _Name [])    { strcpy (Name, _Name);       };

                char* getName()                                 { return Name; };

                int getScopeLevel()                             { return ScopeLevel; };

                bool operator == (const KFGListNodeC& other) const
                                                                { return ((strcmp(Name,
other.Name) == 0)
                                                                        & (ScopeLevel
== other.ScopeLevel)); };

                bool operator != (const KFGListNodeC& other) const
                                                                { return !(*this ==
other); };

                bool operator < (const KFGListNodeC& other) const
                                                                { return (strcmp(Name,
other.Name) < 0); };

                bool operator > (const KFGListNodeC& other) const
                                                                { return (strcmp(Name,
other.Name) > 0); };

                friend ostream& operator << ( ostream& os, const KFGListNodeC&
Node);
        };

#endif

//Class for a list from STL. In this context, KFGListeC inherits all properties
//from the list created using STL list<KFGNodeC>.
#ifdef _KFGProgListHeader
#else
#define _KFGProgListHeader

#include <string>
#include <iostream>
#include <stdio.h>
#include <list>
#include <fstream>
#include <iterator>
#include "KFGLineNode.h"

using namespace std;

class KFGProgListC :  public list<KFGLineNodeC> {

public:
        KFGProgListC() {};

        //void KFGListeAusgeben();

        //void KFGTokenListToKFGList ();
```

```
};

#endif

//Class for a list from STL. In this context, KFGListeC inherits all properties

//from the list created using STL list<KFGNodeC>.

#ifdef _KFGTokenListHeader

#else

#define _KFGTokenListHeader



#include <string>
#include <iostream>
#include <stdio.h>
#include <list>
#include <fstream>
#include <iterator>
#include "KFGTokenNode.h"

using namespace std;


class KFGTokenListC :  public list<KFGTokenNodeC> {

public:

        KFGTokenListC() {};

        void KFGListeAusgeben();

        //void KFGTokenListToKFGList ();

};

#endif


////////////////////////////////////////////////////////////////////////////////
///
// Class for auxiliary list in order to produce a CFG therefrom.
// This class satisfies the nice-demands for the STL;
// i.e. the following is true for a class T:
// 0. It supports the copy constructor T (const T&),
// 1. the assignment operator T& operator= (const T&),
// 2. the comparison operator bool operator== (const T, &const T&) and
// 3. the unequal-to operator!=
// in such a way that:
// (a) { TRUE } T a(b) {a == b}
// (b) { TRUE } a = b  {a == b}
// (c) { a == a}
// (d) { a == b <==> b == a }
// (e) { (a == b) AND ( b == c ) ==> (a == c) }
// (f) { a != b <==> NOT ( a== b) };
//
//      In addition, all functions, in particular getName, are equality preserving,
        i.e.
// (g) { a == b  == > a.getName() == b.getName() }
//
// (C) 1997 Siemens AG
//
//
////////////////////////////////////////////////////////////////////////////////
// 1997-08-25  Andreas Steinhorst     -
////////////////////////////////////////////////////////////////////////////////
#ifdef _KFGTokenNodeHeader
#else
#define _KFGTokenNodeHeader
```

```
#include <string>
#include <iostream>

using namespace std;

        typedef char StringT [256];
        typedef enum
{N,PL,IF,BT,ET,BE,EE,FOR,BF,EF,WHILE,BW,EW,SW,CA,ECA,BRK,RET,DEFA,EDEFA,DO,BDOW,DOW
S,NL,DEF,BFUNC,EFUNC,TD,OUT,FOR_D,PF} TokenNodeTypeT;

        class KFGTokenNodeC {
        private:

                StringT Name;
                TokenNodeTypeT TokenNodeType;
                int ScopeLevel;
                int LineNumber;

        public:

                KFGTokenNodeC()                         { strcpy (Name, "-- unknown --
");
                                                                ScopeLevel = 0;

TokenNodeType = N;
                                                                LineNumber =
0; };

                KFGTokenNodeC(char _Name [], TokenNodeTypeT _TokenNodeType)
                                                                { strcpy (Name,
_Name);
                                                              ScopeLevel = 0;
                                                                 TokenNodeType
= _TokenNodeType;
                                                             LineNumber = 0;};

                KFGTokenNodeC(char _Name [], int _ScopeLevel, TokenNodeTypeT
_TokenNodeType)
                                                                { strcpy (Name,
_Name);
                                                              ScopeLevel =
_ScopeLevel;
                                                             TokenNodeType =
_TokenNodeType;
                                                             LineNumber = 0; };

                KFGTokenNodeC(char _Name [], int _ScopeLevel, TokenNodeTypeT
_TokenNodeType, int _LineNumber)
                                                                { strcpy (Name,
_Name);
                                                              ScopeLevel =
_ScopeLevel;
                                                                 TokenNodeType
= _TokenNodeType;
                                                                 LineNumber =
_LineNumber; };

                void setName(char _Name [])    { strcpy (Name, _Name);       };

                char* getName()                                { return Name; };

                int getScopeLevel()                            { return ScopeLevel; };

                int getLineNumber()                            { return LineNumber; };

                TokenNodeTypeT getTokenNodeType()     { return TokenNodeType; };

                bool operator == (const KFGTokenNodeC& other) const
                                                                { return
((strcmp(Name, other.Name) == 0)
                                                                  & (ScopeLevel
== other.ScopeLevel)
                                                               & (TokenNode
Type == other.TokenNodeType)); };

                bool operator != (const KFGTokenNodeC& other) const
```

```
                                                    { return !(*this ==
other); };
                bool operator < (const KFGTokenNodeC& other) const
                                                                    { return
(strcmp(Name, other.Name) < 0); };
                bool operator > (const KFGTokenNodeC& other) const
                                                                    { return
(strcmp(Name, other.Name) > 0); };
                friend ostream& operator << ( ostream& os, const KFGTokenNodeC&
Node);
        };

#endif

#ifdef _KFGUseHeader

#else

#define _KFGUseHeader


#include <string>
#include <iostream>

using namespace std;

        typedef char StringT [256];


        class KFGUseC {

        private:

                StringT Use;

                int ScopeLevelU;

        public:

                KFGUseC()                       { strcpy (Use, "-- unknown --");
                                                  ScopeLevelU = 0; };

                KFGUseC(char _Use [], int _ScopeLevelU)
                                                { strcpy (Use, _Use);
                                                        ScopeLevelU =
_ScopeLevelU; };
                void setUse(char _Use [])       { strcpy (Use, _Use); };

                char* getUse()                          { return Use;  };

                int getScopeLevelU()            { return ScopeLevelU; };

                bool operator == (const KFGUseC& other) const
                                                { return ((strcmp(Use,
other.Use) == 0)
                                                        & (ScopeLevelU
== other.ScopeLevelU)); };
                bool operator != (const KFGUseC& other) const
                                                { return !(*this ==
other); };
```

```
                bool operator < (const KFGUseC& other) const
                                                        { return (strcmp(Use,
other.Use) < 0); };
                bool operator > (const KFGUseC& other) const
                                                        { return
(strcmp(Use, other.Use) > 0); };
                friend ostream& operator << ( ostream& os, const KFGUseC& Node);
        };

#endif

#ifndef _uwggraphC
#define _uwggraphC

#include <Graph.h>

#include "uwgknoten.h"
#include "uwgkante.h"

#include "AS_Slice.h"

using namespace std;

class uwggraphC : public Graph < uwgknotenC, uwgkanteC > {

public:
                short int folded;

                uwggraphC() {};

                SliceC::iterator findOutputNode(SliceC & S1);

                SliceC::iterator defineIterator(SliceC & S1, int SliceNr);

                void addFirstNodesToFT(SliceC & S1, int KnotenNr, int Pos1);

                SliceC::iterator returnCondNode(SliceC & S1, int SliceNr);

                int checkUWGNode(int SliceNr);

                void SliceAusgeben(SliceC & S1);

                void startBuildFT(SliceC & S1);

                void buildInLoopTree(SliceC &  S1, int Pos1, SliceC::iterator
SLPUSE, SliceC::iterator SLNODE);

                void buildIn_D1_Part(SliceC & S1, int posOR_D1, SliceC::iterator
SLPUSE, SliceC::iterator SLNODE, int D2_RefNr);

                SliceC::iterator findOutmostPUse(SliceC & S1, SliceC::iterator
SLPUSE);

                int checkPUses1(SliceC & S1, SliceC::iterator SLPUSE,
SliceC::iterator SLPUSEREF);

                int checkPUses2(SliceC & S1, SliceC::iterator SLPUSE,
SliceC::iterator SLPUSEREF);

                SliceC::iterator find_D1_Node(SliceC &  S1, SliceC::iterator SLPUSE,
SliceC::iterator SLNODE);

                void addAllD1Nodes(SliceC & S1, SliceC::iterator SLNODE, int
posGatter, int D2_RefNr);

                SliceC::iterator findLastPUse(SliceC & S1, int SliceNr);

                SliceC::iterator findLastPUse2(SliceC & S1, int SliceNr, int RefNr);
```

```
                void checkLoopOrCond(SliceC & S1, SliceC::iterator SL1,
SliceC::iterator SLORIG, int Posi1);

                void buildIn_D2_Part(SliceC & S1, int posOR_D2, SliceC::iterator
SL1, SliceC::iterator SLPUSE, int KnotenNrD2);

                int findD2Node(SliceC & S1, SliceC::iterator PUSE);

                int buildInIfTree(SliceC & S1, SliceC::iterator SLIF,
SliceC::iterator SLORIG, int pos1);

                SliceC::iterator lookForNextPUse(SliceC & S1, SliceC::iterator
SNODE, SliceC::iterator SLPUSE);

                void build_IFKF_Part(SliceC & S1, int posKFOR, SliceC::iterator
SLPUSE);

                void build_A1_Part(SliceC & S1, int posIFDFOR, SliceC::iterator
SLIF, SliceC::iterator SLORIG);

                void buildD2_IFKF_Part(SliceC & S1, int posKFOR, SliceC::iterator
SLPUSE);

                void buildD2_A1_Part(SliceC & S1, int posIFDFOR, SliceC::iterator
SLIF, SliceC::iterator SLNODE);

                int buildD2InIfTree(SliceC & S1, SliceC::iterator SLIF,
SliceC::iterator SLORIG, int pos1);

                void buildLoopKF_Part(SliceC & S1, int posAND_KF, int posLOOP_KF,
int posD2, SliceC::iterator SLOOP);

                int buildIn_LoopKF_ANDPart(SliceC & S1, int posAND_KF,
SliceC::iterator SLPUSE);

                void buildIn_LoopKF_ORPart(SliceC & S1, int posAND_KF,
SliceC::iterator SL1, int KnotenNrD2);

};

#endif

#ifndef _uwgkanteC
#define _uwgkanteC

#include <iostream>

using namespace std;

class uwgkanteC {

public:
                short int folded;
                int Number;

                uwgkanteC()                                          {};

                uwgkanteC(int _Number)              { Number = _Number; };

                bool operator == (const uwgkanteC& other) const
                                        { return (Number == other.Number); };

                bool operator != (const uwgkanteC& other) const
                                        { return !(*this == other);   };

                bool operator < (const uwgkanteC& other) const
                                                { return (Number ==
other.Number); };

                bool operator > (const uwgkanteC& other) const
                                        { return (Number == other.Number); };

        friend ostream& operator << ( ostream& os, const uwgkanteC& Node);
};

#endif
```

```
#ifndef _uwgknotenC
#define _uwgknotenC

#include <iostream>

const int maxKnotentextLength=128;
const int maxBemerkLength=1000;
const int maxWahrVarLength=16;

typedef struct ww {
        short int Wert;
        struct ww *next;
        } TMCSWert;


typedef struct el {
        struct ftgatter* unabElement;
        struct ww *next;
        struct el *nextel;
        } TMinSet;

typedef enum {AND, OR, CAUSE, EFFECT, NOT} NodeIdentT;

using namespace std;

class uwgknotenC {

public:
                static int DummyNr;
                int GatterIdent;
                int outof;
                int CauseNr;
                char gattertext[maxKnotentextLength];
        char gatterbemerkung[maxBemerkLength];
                double wahrsch;
                char wahrvar[maxWahrVarLength];
                NodeIdentT NodeIdent;

                uwgknotenC()                                    { NodeIdent = EFFECT;

CauseNr = 0;                                                      strcpy (gattertext,

"-- unknown --");                                                 strcpy (gatter-

bemerkung, "--none--");

GatterIdent = DummyNr++; };

                uwgknotenC(NodeIdentT _NodeIdent)                                  {

NodeIdent = _NodeIdent;

CauseNr = 0;

strcpy(gattertext, "-- unknown --");

strcpy(gatterbemerkung, "--none--");

GatterIdent = DummyNr++; };

                uwgknotenC(NodeIdentT _NodeIdent, int _CauseNr/*, char
_gattertext[]*/)                                                   {

NodeIdent =  _NodeIdent;

CauseNr = _CauseNr;

strcpy(gattertext, "-- unknown --");

strcpy(gatterbemerkung, "--none--");

/*strcpy(gattertext, _gattertext);*/                                GatterIdent =

DummyNr++; };
```

```
                uwgknotenC(NodeIdentT _NodeIdent, int _CauseNr, char _gattertext[],
char _gatterbemerkung[])                                                { NodeIdent =
_NodeIdent;                                                               CauseNr =
_CauseNr;
strcpy(gattertext, "-- unknown --");
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, _gatterbemerkung);                                GatterIdent =
DummyNr++; };
                uwgknotenC(NodeIdentT _NodeIdent, int _CauseNr, char _gattertext[])
                                                                        { NodeIdent =
_NodeIdent;                                                               CauseNr =
_CauseNr;                                                                 /*strcpy
(gattertext, "-- unknown --");*/
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, "--none--");                                      GatterIdent =
DummyNr++; };
                uwgknotenC(NodeIdentT _NodeIdent, char _gattertext[], char
_gatterbemerkung[])                                                     { NodeIdent =
_NodeIdent;                                                               CauseNr = 0;
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, _gatterbemerkung);                                GatterIdent =
DummyNr++; };
                uwgknotenC(NodeIdentT _NodeIdent, char _gattertext[])
                                                                        { NodeIdent =
_NodeIdent;                                                               CauseNr = 0;
strcpy(gattertext, _gattertext);
strcpy(gatterbemerkung, "--none--");                                      GatterIdent =
DummyNr++; };
                int getCauseNr()                        { return CauseNr; };
                int getGatterIdent()            { return GatterIdent; };
                NodeIdentT getNodeIdent() const         { return NodeIdent; };
                bool operator == (const uwgknotenC& other) const
                                                                        { return ((CauseNr ==
other.CauseNr)                                                                    &
(NodeIdent == other.NodeIdent)                                                    &
(strcmp(gattertext,other.gattertext) == 0)                                /*& (GatterIdent ==
other.GatterIdent)*/); };
                bool operator != (const uwgknotenC& other) const
                                                                        { return !(*this ==
other); };
                bool operator < (const uwgknotenC& other) const
                                                                        { return (GatterIdent <
other.GatterIdent); };
                bool operator > (const uwgknotenC& other) const
```

```
                                                                    { return
(GatterIdent > other.GatterIdent); };

                friend ostream& operator << ( ostream& os, const uwgknotenC& Node);

};

#endif

#include "AS_GraphKante.h"
#include <iostream>

        ostream& operator << ( ostream& os, const GraphKanteC & Kante){
                os << Kante.KantenTyp;
                return os;
        }

#include "AS_GraphNode.h"

#include <iostream>

        ostream& operator << ( ostream& os, const GraphNodeC& Node){
                os << endl << Node.NodeNr;
                return os;
        }


#include "AS_Slice.h"

#include <iostream>

#include <iterator>

#include <algorithm>

#include "uwggraph.h"

using namespace std;


//F0: Function for selecting the variable for which an error tree is to be created.

//    A pointer to the selected node is returned.
KFGListC::iterator SliceC::defineVariableToSlice(KFGListC & L2) {
        int Number = 0;
        KFGListC::iterator KFGI = L2.begin();
        KFGListC::iterator VARDEFI = L2.begin();
        KFGUseListC::iterator USEIT;
        while ( KFGI != L2.end()) {
                USEIT = KFGI->Uses.begin();
                if ( KFGI->getKFGNodeType() == OP ) {
                        cout << KFGI->getKnotenNummer() << ": " << USEIT->getUse()
<< endl;;
                }
                KFGI++;
        }
        cout << "Enter the node number and press RETURN: ";
        cin >> Number;
        while ( KFGI != L2.begin()) {
                if (KFGI->getKnotenNummer() == Number) {
                        VARDEFI = KFGI;
                }
                KFGI--;
        }
        if (VARDEFI->getKFGNodeType() == OP) {
                cout << "The selected node was No: "  << VARDEFI->getKnotenNummer()
<< endl;
```

```
                return VARDEFI;
        }
        else {
                VARDEFI = L2.begin();
                cout << "This node is not an output node: the program is being
terminated" << endl;
                exit(1);
                return VARDEFI;
        }
}

//F1: Looks for P-Use node, adds it to the slice and draws a KFK.
//    input: iterator for the C-Use or D-Use, CFG list;
void SliceC::sliceForLoops(KFGListC & L2, KFGListC::iterator LOOPIT) {
        int HLevel;
        int check = 0;
        int KnotenNummer = 0;
        KFGListC::iterator HELPIT = LOOPIT;
        while ((LOOPIT->getKFGNodeType() != LC && LOOPIT->getKFGNodeType() != IFC)
|| (LOOPIT->getLevel() != HELPIT->getLevel())) {
                LOOPIT--;
        }
        KnotenNummer = LOOPIT->getNodeNr();
        check = checkForNodes(KnotenNummer);
        //insert(*LOOPIT,*HELPIT,KFK);
        insert(*HELPIT,*LOOPIT,KFK); //the reverse direction of the edge so that
each node knows its predecessor.
        //cout << "eingefügterKnotenF1a: " << LOOPIT->getNodeNr() << endl;
        //cout << "eingefügterKnotenF1a: " << HELPIT->getNodeNr() << endl << endl;
        if (check == 0) {
                findAllDefs(L2, LOOPIT);
        }
        HLevel = LOOPIT->getLevel();
        while (LOOPIT->getLevel() > 2) {
                check == 0;
                HELPIT = LOOPIT;
                while ((LOOPIT->getKFGNodeType() != LC && LOOPIT->getKFGNodeType()
!= IFC) || (LOOPIT->getLevel() >= HELPIT->getLevel())) {
                LOOPIT--;
                }
                KnotenNummer = LOOPIT->getNodeNr();
                check = checkForNodes(KnotenNummer);
                //insert(*LOOPIT,*HELPIT,KFK);
                insert(*HELPIT,*LOOPIT,KFK); //the reverse direction of the edge so
that each node knows its predecessor.
                //cout << "eingefügterKnotenF1b: " << LOOPIT->getNodeNr() << endl;
                //cout << "eingefügterKnotenF1b: " << HELPIT->getNodeNr() << endl <<
endl;
                if (check == 0) {
                        findAllDefs(L2, LOOPIT);
                }
                HLevel = LOOPIT->getLevel();
        }
}

//F2: Looks for all D-Uses for a P-Use, adds them to the slice and draws DFK.
//    input: iterator for the node with the P-Use list, CFG list;
void SliceC::findAllDefs(KFGListC & L1, KFGListC::iterator ITER) {
        int AktLevel;
        int dummyIf = 0;
        int KnotenNummerEnde = 0;
        AktLevel = ITER->getLevel();
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPIF = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator UPPERLIMIT = ITER;
        KFGP_UseListC::iterator PUSEIT;
        PUSEIT = ITER->P_Uses.begin();
        KFGDefListC::iterator DEFIT1;
        if (ITER->getKFGNodeType() == LC) {
                while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP->getLevel()
!= AktLevel)) {
                        HELPLOOP++;
                }
                HELPIT = HELPLOOP;
                KnotenNummerEnde = HELPLOOP->getNodeNr();
```

```
        }
        while (PUSEIT != ITER->P_Uses.end()) {
                UPPERLIMIT = findUpperLimit(L1, ITER, PUSEIT);
                HELPLOOP = findLowerLimit(L1, ITER, UPPERLIMIT);
                //insert (*UPPERLIMIT,*ITER,DFK);
                insert(*ITER,*UPPERLIMIT,DFK); //the reverse direction of the edge
so that each node knows its predecessor.
                //cout << "eingefügterKnotenF2a: " << UPPERLIMIT->getNodeNr() <<
endl;
                //cout << "eingefügterKnotenF2a: " << ITER->getNodeNr() << endl;
                while (HELPLOOP != UPPERLIMIT) {
                        HELPLOOP--;
                        if (HELPLOOP->getKFGNodeType() == EEl) {
                                dummyIf = 1;
                        }
                        if ((HELPLOOP->getKFGNodeType() == ETh) && (HELPLOOP-
>getLevel() < ITER->getLevel()) && (dummyIf == 0)) {
                                HELPIF = HELPLOOP;
                                while ((HELPLOOP->getKFGNodeType() != IFC) ||
(HELPLOOP->getLevel() != HELPIF->getLevel())) {
                                        HELPLOOP--;
                                }
                                dummyIf = 0;
                        }
                        if (HELPLOOP->getKFGNodeType() == NO) {
                                DEFIT1 = HELPLOOP->Defs.begin();
                                while (DEFIT1 != HELPLOOP->Defs.end()) {
                                if (strcmp(PUSEIT->getUse(),DEFIT1->getDef()) == 0) {
                                        //insert(*HELPLOOP,*ITER,DFK);
                                        insert(*ITER,*HELPLOOP,DFK); //the
reverse direction of the edge so that each node knows its predecessor.
                                        defInLoop(L1,HELPLOOP);
                                        //cout << "eingefügterKnotenF2b: " <<
HELPLOOP->getNodeNr() << endl;
                                        //cout << "eingefügterKnotenF2b: " <<
ITER->getNodeNr() << endl << endl;
                                        if (!HELPLOOP->Uses.empty()) {
                                                findDefToCUse(L1, HELPLOOP);
                                        }
                                }
                                DEFIT1++;
                                }
                        }
                }
                PUSEIT++;
        }
        KFKPUseToCUse(ITER,KnotenNummerEnde);
}

//F5: Function looks for the first def above a p-use or c-use and returns an
//    iterator for this upper limit when looking for all defs.
KFGListC::iterator SliceC::findUpperLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGP_UseListC::iterator PUSEIT) {
        int dummy = 0;
        int dummy1 = 0;
        int dummyIf = 0;
        int AktLevel = 0;
        KFGListC::iterator HELPDEF = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator HELPIF = ITER;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPDEFRETURN = ITER;
        KFGDefListC::iterator DEFIT;
        AktLevel = ITER->getLevel();
        //find the first def above the p-use.
        while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                HELPDEF--;
                if (HELPLOOP->getKFGNodeType() == EEl) {
                        dummyIf = 1;
                }
                if ((HELPDEF->getKFGNodeType() == ETh) && (HELPDEF->getLevel() <
ITER->getLevel()) && (dummyIf == 0)) {
                        HELPIF = HELPDEF;
                        while ((HELPDEF->getKFGNodeType() != IFC) || (HELPDEF-
>getLevel() != HELPIF->getLevel())) {
                                HELPDEF--;
                        }
```

```
                        dummyIf = 0;
                }
                if (HELPDEF->getKFGNodeType() == NO) {
                        DEFIT = HELPDEF->Defs.begin();
                        if (strcmp(DEFIT->getDef(),PUSEIT->getUse()) == 0) {
                                HELPDEFRETURN = HELPDEF;
                                dummy = 1;
                        }
                }
        }
        if ((HELPDEF == L1.begin()) && (HELPDEF->getKFGNodeType() == NO)) {
                DEFIT = HELPDEF->Defs.begin();
                if (strcmp(DEFIT->getDef(),PUSEIT->getUse()) == 0) {
                        HELPDEFRETURN = HELPDEF;
                        dummy = 1;
                }
        }
        //find the outermost loop if it actually exists.
        if (ITER->getLevel() > 2) {
                while ((HELPIT != L1.end()) && (dummy1 != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel()
< AktLevel)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        AktLevel = HELPIT->getLevel();
                                }
                                else {
                                        dummy1 = 1;
                                }
                        }
                }
                HELPIT = HELPLOOP;
                //go to the start of the outermost loop:
                while ((HELPIT->getKFGNodeType() != LC) || (HELPIT->getLevel() !=
HELPLOOP->getLevel())) {
                        HELPIT--;
                }
                //if the def node found is outside the outermost loop and
                //its ScopeLevel is greater than that for the outermost loop, look
                //for whether there is still another def node further up.
                if (HELPDEF->getNodeNr() > HELPIT->getNodeNr()) {
                        if (HELPDEF->getLevel() >= HELPIT->getLevel()) {
                                dummy = 0;
                                while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                                        HELPDEF--;
                                        DEFIT = HELPDEF->Defs.begin();
                                        if (strcmp(DEFIT->getDef(),PUSEIT->getUse())
== 0) {
                                                HELPDEFRETURN = HELPDEF;
                                                dummy = 1;
                                        }
                                }
                        }
                }
        }
        return HELPDEFRETURN;
}

//F6: Function looks, on the basis of the upper end, for the lower end of the
//    range by needing to look for the defs.
//       An iterator for the lower limit is returned.
KFGListC::iterator SliceC::findLowerLimit(KFGListC & L1, KFGListC::iterator ITER,
KFGListC::iterator UPPERLIMIT) {
        int dummy1 = 0;
        int AktLevel = 0;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator LOWERLIMIT = ITER;
        AktLevel = ITER->getLevel();
        if (ITER->getKFGNodeType() == LC) {
                while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP->getLevel()
!= AktLevel)) {
                        HELPLOOP++;
                }
```

```
}
//find the end of the outermost loop.
```

```
        if (ITER->getLevel() > 2) {
                while ((HELPIT != L1.end()) && (dummy1 != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel()
5   < AktLevel)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        AktLevel = HELPIT->getLevel();
                                }
10                              else {
                                        dummy1 = 1;
                                }
                        }
                }
15              HELPIT = HELPLOOP;                      //HELPLOOP points to the end
    of the outermost loop, otherwise it points to the p-use.
                while ((HELPIT->getKFGNodeType() != LC) || (HELPIT->getLevel() !=
    HELPLOOP->getLevel())) {
                        HELPIT--;
20              }                                                       //HELPIT points
    to the start of the outermost loop.
                if (HELPIT->getNodeNr() < UPPERLIMIT->getNodeNr()) {
                        LOWERLIMIT = HELPLOOP; //if the upper limit is outside the
    loops.
25              }
                else {
                        AktLevel = UPPERLIMIT->getLevel();
                        while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP-
    >getLevel() <= AktLevel)) {
30                              HELPLOOP--;
                        }
                        LOWERLIMIT = HELPLOOP; //if the upper limit is inside the
    loop.
                }
35      }
        else {
                LOWERLIMIT = HELPLOOP;
        }
        return LOWERLIMIT;
40  }

    //F3: Function draws a KFK from the P-Use to all C-Uses which are at the same
    //    loop or branch level.
    //    input: iterator for P-Use node.
45  void SliceC::KFKPUseToCUse(KFGListC::iterator PUSEIT, int SchleifenEnde) {
        SliceC::iterator SL1 = begin();
        SliceC::iterator SLHELP = begin();
        KFGListC::iterator ITER = PUSEIT;
        KFGUseListC::iterator USEIT;
50      KFGP_UseListC::iterator PUSEIT2;
        KFGUseListC::iterator CUSEIT2;
        int KnotenLevel;
        int KnotenNummer;
        KnotenNummer = PUSEIT->getNodeNr();
55      KnotenLevel = PUSEIT->getLevel();
        while ((*SL1).first.getNodeNr() != KnotenNummer)
                SL1++;
        SLHELP = SL1;
        SL1 = begin();
60      while (SL1 != end()) {
                if ((KnotenNummer > (*SL1).first.getNodeNr()) && (SchleifenEnde <
    (*SL1).first.getNodeNr())) {
                        if (((*SL1).first.getKFGNodeType() == NO) &&
    ((*SL1).first.getLevel() == KnotenLevel))  {
65                              //insert((*SLHELP).first,(*SL1).first,KFK);
                                insert((*SL1).first,(*SLHELP).first,KFK); //the
    reverse direction of the edge so that each node knows its predecessor.
                                //cout << "eingefügterKnotenF3: " <<
    (*SL1).first.getNodeNr() << endl;
70                              //cout << "eingefügterKnotenF3: " <<
    (*SLHELP).first.getNodeNr() << endl;
                        }
                }
                SL1++;
75      }
    }
```

```
//F4: If the node also contains c-uses in addition to the defs, function looks for
//    all defs for these c-uses and draws DFK from the def found to the c-use.
void SliceC::findDefToCUse(KFGListC & L1, KFGListC::iterator ITER) {
        int HLevel;
        int check = 0;
        int dummy = 0;
        int dummyIf = 0;
        int KnotenNummer = 0;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator HELPIF = ITER;
        KFGListC::iterator LOWERLIMIT = ITER;
        KFGListC::iterator UPPERLIMIT = ITER;
        KFGDefListC::iterator DEFIT1;
        KFGUseListC::iterator USEIT1;
        USEIT1 = ITER->Uses.begin();
        HLevel = ITER->getLevel();
        while (USEIT1 != ITER->Uses.end()) {
                UPPERLIMIT = findUpperLimitFromCUse(L1, ITER, USEIT1);
                HELPIT = findLowerLimitFromCUse(L1, ITER, UPPERLIMIT);
                UPPERLIMIT--;
                //cout << "eingefügterKnotenF4a: " << UPPERLIMIT->getNodeNr() <<
endl;
                //cout << "eingefügterKnotenF4a: " << ITER->getNodeNr() << endl;
                while (HELPIT != UPPERLIMIT) {
                        if (HELPLOOP->getKFGNodeType() == EEl) {
                                dummyIf = 1;
                        }
                        if ((HELPIT->getKFGNodeType() == ETh) && (HELPIT->getLevel()
<= ITER->getLevel()) && (dummyIf == 0)) {
                                HELPIF = HELPIT;
                                while ((HELPIT->getKFGNodeType() != IFC) || (HELPIT-
>getLevel() != HELPIF->getLevel())) {
                                        HELPIT--;
                                }
                                dummyIf = 0;
                        }
                        if (HELPIT->getKFGNodeType() == NO) {
                                DEFIT1 = HELPIT->Defs.begin();
                                if (strcmp(DEFIT1->getDef(),USEIT1->getUse()) == 0) {
                                        if (!HELPIT->Uses.empty()) {
                                                KnotenNummer = HELPIT->getNodeNr();
                                                check = checkForNodes(KnotenNummer);
                                        }
                                        //insert(*HELPIT,*ITER,DFK);
                                        insert(*ITER,*HELPIT,DFK); //the reverse
direction of the edge so that each node knows its predecessor.
                                        defInLoop(L1,HELPIT);
                                        //cout << "eingefügterKnotenF4b: " << HELPIT-
>getNodeNr() << endl;
                                        //cout << "eingefügterKnotenF4b: " << ITER-
>getNodeNr() << endl << endl;
                                        if (check == 0) {
                                                findDefToCUse(L1,HELPIT);
                                        }
                                }
                        }
                        HELPIT--;
                }
                USEIT1++;
        }
}


//F8: Function looks for the lower limit from which the defs for a c-use can
//    be sought.
KFGListC::iterator SliceC::findLowerLimitFromCUse(KFGListC & L1, KFGListC::iterator
ITER, KFGListC::iterator UPPERLIMIT) {
        int HLevel;
        int dummy = 0;
        KFGListC::iterator HELPIT = ITER;
        KFGListC::iterator HELPLOOP = ITER;
        KFGListC::iterator LOWERLIMIT = ITER;
        KFGDefListC::iterator DEFIT1;
        KFGUseListC::iterator USEIT1;
        USEIT1 = ITER->Uses.begin();
```

```
        HLevel = ITER->getLevel();
        //Is the c-use node in a loop?
        if (ITER->getLevel() >= 2) {
                while ((HELPIT != L1.end()) && (dummy != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel()
<= HLevel)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        HLevel--;
                                }
                                else {
                                        dummy = 1;
                                }
                        }
                }
        }
        //The iterator HELPLOOP now points to the end of the outermost loop,
surrounding the node,
        //if this loop exists, otherwise the iterator points to the c-use node
itself.
        if (HELPLOOP != ITER) {
                HELPIT = HELPLOOP;              //Iterator points to the end of the
outermost loop.
                while ((HELPIT->getKFGNodeType() != LC) || (HELPIT->getLevel() !=
HELPLOOP->getLevel())) {
                        HELPIT--;
                }
                if (HELPIT->getNodeNr() < UPPERLIMIT->getNodeNr()) {
                        LOWERLIMIT = HELPLOOP; //upper limit outside the outermost
loop;
                }
                else if (ITER->getLevel() == UPPERLIMIT->getLevel()) {
                        LOWERLIMIT = ITER;              //upper limit with the c-use
in the same loop;
                }
                else if (ITER->getLevel() < UPPERLIMIT->getLevel()) {
                        LOWERLIMIT = HELPLOOP;  //upper limit in the same outer loop
but
                }                                                       //inside in
another loop, which is before the c-use;
                else {
                        HLevel = UPPERLIMIT->getLevel();
                        while ((HELPLOOP->getKFGNodeType() != EL) || (HELPLOOP-
>getLevel() <= HLevel)) {
                                HELPLOOP--;
                        }
                        LOWERLIMIT = HELPLOOP; //upper limit in a lower loop than the
c-use;
                }
        }
        else {
                LOWERLIMIT = ITER;              //no outer loop present.
        return LOWERLIMIT;
}

//F7: Function looks for the upper limit up to which defs for a c-use can be
sought. This
//    works in a similar way to the corresponding function for p-uses.
KFGListC::iterator SliceC::findUpperLimitFromCUse(KFGListC & L1, KFGListC::iterator
ITER1, KFGUseListC::iterator USEIT) {
        int dummy = 0;
        int dummy1 = 0;
        int dummyIf = 0;
        int AktLevel = 0;
        KFGListC::iterator HELPDEF = ITER1;
        KFGListC::iterator HELPLOOP = ITER1;
        KFGListC::iterator HELPIT = ITER1;
        KFGListC::iterator HELPIF = ITER1;
        KFGListC::iterator HELPDEFRETURN = ITER1;
        KFGDefListC::iterator DEFIT;
        AktLevel = ITER1->getLevel();
        //cout << "processing node: " << ITER1->getNodeNr() << endl;
        //find the first def above the use.
        do {
                if (HELPLOOP->getKFGNodeType() == EE1) {
                        dummyIf = 1;
```

```
                }
                if ((HELPDEF->getKFGNodeType() == ETh) && (HELPDEF->getLevel() <=
ITER1->getLevel()) && (dummyIf == 0)) {
                        HELPIF = HELPDEF;
                        while ((HELPDEF->getKFGNodeType() != IFC) || (HELPDEF-
>getLevel() != HELPIF->getLevel())) {
                                HELPDEF--;
                        }
                        dummyIf = 0;
                }
                if (HELPDEF->getKFGNodeType() == NO) {
                        DEFIT = HELPDEF->Defs.begin();
                        if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                                HELPDEFRETURN = HELPDEF;
                                if (HELPDEF != ITER1) {
                                        dummy = 1;
                                }
                        }
                }
                HELPDEF--;
        } while ((HELPDEF != L1.begin()) && (dummy != 1));
        if ((HELPDEF == L1.begin()) && (HELPDEF->getKFGNodeType() == NO)) {
                DEFIT = HELPDEF->Defs.begin();
                if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                        HELPDEFRETURN = HELPDEF;
                }
        }
        HELPDEF = HELPDEFRETURN;
        //cout << "first def: " << HELPDEFRETURN->getNodeNr() << endl;
        //find the outermost loop if it actually exists.
        if (ITER1->getLevel() >= 2) {
                while ((HELPIT != L1.end()) && (dummy1 != 1)) {
                        HELPIT++;
                        if ((HELPIT->getKFGNodeType() == EL) && (HELPIT->getLevel()
<= AktLevel)) {
                                HELPLOOP = HELPIT;
                                if (HELPIT->getLevel() > 2) {
                                        AktLevel--;
                                }
                                else {
                                        dummy1 = 1;
                                }
                        }
                }
                HELPIT = HELPLOOP;
                //cout << "End of the outermost loop: " << HELPIT->getNodeNr() <<
endl;
                //go to the start of the outermost loop if it exists;
                if (HELPLOOP->getKFGNodeType() == EL) {
                        while ((HELPIT->getKFGNodeType() != LC) || (HELPIT-
>getLevel() != HELPLOOP->getLevel())) {
                                HELPIT--;
                        }
                        //cout << "Start of the outermost loop: " << HELPIT-
>getNodeNr() << endl;
                        //if the def node found is outside the outermost loop and
                        //its ScopeLevel is greater than that for the outermost
loop, look for whether
                        //there is still another def node further up.
                        if (HELPDEF->getNodeNr() > HELPIT->getNodeNr()) {
                                if (HELPDEF->getLevel() >= HELPIT->getLevel()) {
                                        dummy = 0;
                                        while ((HELPDEF != L1.begin()) && (dummy !=
1)) {
                                                HELPDEF--;
                                                DEFIT = HELPDEF->Defs.begin();
                                                if (strcmp(DEFIT->getDef(),USEIT-
>getUse()) == 0) {
                                                        HELPDEFRETURN = HELPDEF;
                                                        dummy = 1;
                                                }
                                        }
                                }
                        }
                        //c-use within the outermost loop;
                        else {
```

```
if (HELPDEF->getLevel() > ITER1->getLevel()) {
        dummy = 0;
        while (HELPDEF != HELPIT) {
```

5

```
                                                HELPDEF--;
                                                DEFIT = HELPDEF->Defs.begin();
                                                if (strcmp(DEFIT->getDef(),USEIT-
>getUse()) == 0) {
                                                        HELPDEFRETURN = HELPDEF;
                                                }
                                        }
                                }
                        }
                        //c-use not in a loop, but in a branch;
                        else {
                                if (HELPDEF->getLevel() > ITER1->getLevel()) {
                                        dummy = 0;
                                        while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                                                HELPDEF--;
                                                DEFIT = HELPDEF->Defs.begin();
                                                if (strcmp(DEFIT->getDef(),USEIT->getUse())
== 0) {
                                                        HELPDEFRETURN = HELPDEF;
                                                        dummy = 1;
                                                }
                                        }
                                }
                        }
                }
                //c-use not in a loop;
                else {
                        if (HELPDEF->getLevel() > ITER1->getLevel()) {
                                dummy = 0;
                                while ((HELPDEF != L1.begin()) && (dummy != 1)) {
                                        HELPDEF--;
                                        DEFIT = HELPDEF->Defs.begin();
                                        if (strcmp(DEFIT->getDef(),USEIT->getUse()) == 0) {
                                                HELPDEFRETURN = HELPDEF;
                                                dummy = 1;
                                        }
                                }
                        }
                }
                //cout << "UpperLimitFunktion: " << HELPDEFRETURN->getNodeNr() << endl;
                return HELPDEFRETURN;
        }

        //F9: Function checks whether there is already a node in
              the slice.
              If there is, a one is returned, otherwise a zero is returned.
        // input: node number of the node sought;
        // output: integer value 0 or 1;
        int SliceC::checkForNodes(int NodeNumber) {
                int dummy = 0;
                SliceC::iterator SL1 = begin();
                while ((SL1 != end()) && (dummy != 1)) {
                        if ((*SL1).first.getNodeNr() == NodeNumber) {
                                dummy = 1;
                        }
                        SL1++;
                }
                if (dummy == 1) {
                        return 1;
                }
                else {
                        return 0;
                }
        }

        //F10: Function examines whether there is a def in a loop; if there is,
        //     function F1, which looks for the p-use, is called. This first checks whether
              the p-use found is already present in the slice.
        void SliceC::defInLoop(KFGListC & L1, KFGListC::iterator ITER) {
                int dummyLoop = 0;
                KFGListC::iterator HELPIT = ITER;
                while ((HELPIT != L1.begin()) && (dummyLoop != 1)) {
                        HELPIT--;
                        if (((HELPIT->getKFGNodeType() == LC) || (HELPIT->getKFGNodeType()
```

```
                    == IFC))
    && (HELPIT->getLevel() == ITER->getLevel())) {
                            sliceForLoops(L1,ITER);
5
```

```
                dummyLoop = 1;
                }
        }
}


//F11: The selected start node for the slice is transferred to the function,
//     and the function investigates whether this node is in a loop; if it is,
//     the function "sliceforloops" is first called, otherwise "findDefToCUse" is
       called immediately.
void SliceC::startBuildSlice(KFGListC & L1/*, KFGListC::iterator OUTPUTIT*/) {
        int dummyNode = 0;
        KFGListC::iterator STARTVARI = defineVariableToSlice(L1);
        KFGListC::iterator HELPOUT = STARTVARI;
        while ((HELPOUT != L1.begin()) && (dummyNode != 1)) {
                HELPOUT--;
                if (((HELPOUT->getKFGNodeType() == LC) || (HELPOUT->getKFGNodeType()
                == IFC))
        && (HELPOUT->getLevel() == STARTVARI->getLevel())) {
                        sliceForLoops(L1,STARTVARI);
                        dummyNode = 1;
                }
        }
        //cout << "F11" << OUTPUTIT->getNodeNr() << endl;
        findDefToCUse(L1,STARTVARI);
}

void SliceC::sliceAusgeben() {
        ofstream Ziel("Slice2.slc");
        ostream_iterator<KFGListNodeC> POSIT(Ziel, "\n");
        //ostream_iterator<KFGListNodeC> POSIT2(Ziel, "\n");
        SliceC::iterator SLC = begin();
        while (SLC != end()) {
                *POSIT++ = (*SLC).first;
                SliceC::Nachfolger::iterator IT = (*SLC).second.begin();
                SliceC::Nachfolger::iterator ITEND = (*SLC).second.end();
                while (IT != ITEND) {
                        //a = (*IT).first;
                        *POSIT++ = SLC[(*IT).first].first;
                        *IT++;
                }
                ++SLC;
        }
}



#include "KFGDef.h"

#include <iostream>

        ostream& operator << ( ostream& os, const KFGDefC& Node){

                os << "DEF:            " << Node.Def << "     " << Node.ScopeLevelD;

                return os;

        }

#include "KFGLineNode.h"

#include <iostream>

        ostream& operator << ( ostream& os, const KFGLineNodeC& Node){
                os << Node.Name << " " << Node.LineNumber <<  endl;
                return os;
        }


//Function which constructs a CFG from KFGTokenList.
```

```
#include <iostream>

#include "KFGList.h"

#include "KFGTokenList.h"
#include "KFGUse.h"
#include "KFGDef.h"
#include "KFGProgList.h"
#include <algorithm>


void KFGListC::TokenList2KFGList(KFGTokenListC & L1) {
        int dummy;
        KFGTokenListC::iterator TLI = L1.end();
        KFGP_UseListC::iterator P_USEI;
        KFGUseListC::iterator USEI;
        dummy = 0;
        //while (strcmp(TLI->getName(),"main") != 0) {
        while ((TLI != L1.begin()) && (dummy != 1)) {
                if (TLI->getTokenNodeType() == N) {
                        if (strcmp(TLI->getName(),"main") == 0) {
                                dummy = 1;
                        }
                }
                //TLI++;
                switch (TLI->getTokenNodeType()) {
                case PL:
                        {
                                TLI--;
                                KFGListNodeC hN1("NORMAL",NO,TLI->getScopeLevel(),
TLI->getLineNumber());
                                hN1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                                push_front(hN1);
                                TLI--;
                        }
                        break;
                /*case IF:
                        push_front(KFGListNodeC("IF",IFC,TLI->getScopeLevel()));
                        TLI--;
                        break;*/
                case BT:
                        //push_front(KFGListNodeC("BT",BTh,TLI->getScopeLevel()));
                        {
                                TLI--;
                                //KFGP_UseListC::iterator P_USEI;
                                KFGListNodeC hN1("IFCOND",IFC,TLI->getScopeLevel(),
TLI->getLineNumber());
                                while (TLI->getTokenNodeType() != IF) {
                                        KFGUseC hUse1(TLI->getName(),TLI->
getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        //hN1.P_Uses.push_back(KFGUseC(TLI->
getName(),TLI->getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);
                                Entscheidungen++;
                        }
                        break;
                case ET:
                        push_front(KFGListNodeC("ENDTHEN",ETh,TLI->
getScopeLevel()));
                        TLI--;
                        break;
                case BE:
                        push_front(KFGListNodeC("BEGINELSE",BE1,TLI->
getScopeLevel()));
                        TLI--;
                        break;
                case EE:
```

```
                        push_front(KFGListNodeC("ENDELSE",EEl,TLI->
getScopeLevel()));
                        TLI--;
                        break;
5                   /*case WHILE:
                        push_front(KFGListNodeC("WHILE",LC,TLI->getScopeLevel()));
```

```
                        TLI--;
                        break;*/
                case BW:
                        //push_front(KFGListNodeC("BW",BL,TLI->getScopeLevel()));
                        {
                                TLI--;
                                //KFGP_UseListC::iterator P_USEI;
                                KFGListNodeC hN1("LOOPCOND",LC,TLI->getScopeLevel(),
TLI->getLineNumber());
                                while (TLI->getTokenNodeType() != WHILE) {
                                        KFGUseC hUse1(TLI->getName(),TLI->
getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        //hN1.P_Uses.push_back(KFGUseC(TLI->
getName(),TLI->getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);
                                Schleifenentscheidungen++;
                        }
                        break;
                case EW:
                        push_front(KFGListNodeC("ENDLOOP",EL,TLI->getScopeLevel()));
                        TLI--;
                        break;
                case BDOW:
                        {
                                TLI--;
                                KFGListNodeC hN1("DOWHILELOOPCOND",DOWLC,TLI->
getScopeLevel(),TLI->getLineNumber());
                                while (TLI->getTokenNodeType() != DOWS) {
                                        KFGUseC hUse1(TLI->getName(),TLI-
>getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        hN1.P_Uses.push_back(KFGUseC(TLI-
>getName(),TLI->getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);
                        }
                        break;
                case DO:
                        push_front(KFGListNodeC("DOWHILELOOP",DOWL,TLI->
getScopeLevel()));
                        TLI--;
                        Schleifenentscheidungen++;
                        break;
                case EF:
                        {
                                //TLI--;
                                int HLevel;
                                HLevel = TLI->getScopeLevel();
                                push_front(KFGListNodeC("ENDLOOP",EL,TLI->
getScopeLevel()));
                                //while  (TLI->getTokenNodeType() != BF) {
                                while ((TLI->getTokenNodeType() != BF) || (TLI-
>getScopeLevel() != HLevel)) {
                                        TLI--;
                                }
                                TLI--;
                                if (TLI->getTokenNodeType() == PF)
                                        TLI--;
                                KFGListNodeC hN1("NORMAL",NO,TLI->getScopeLevel(),
TLI->getLineNumber());
                                hN1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
```

```
                                Anzahl_Defs++;
                                hN1.Uses.push_back(KFGUseC(TLI->getName(),TLI->
    getScopeLevel()));
                                Anzahl_Uses++;
5                               push_front(hN1);
                                //while (TLI->getTokenNodeType() != EF) {
```

```
                                while ((TLI->getTokenNodeType() != EF) || (TLI-
>getScopeLevel() != HLevel)) {
                                        TLI++;
                                }
                                TLI--;
                                Zaehlschleifenentscheidungen++;
                                Schleifenentscheidungen++;
                        }
                        break;
                case FOR_D:
                        {
                                TLI--;
                                KFGListNodeC hN1("LOOPCOND",LC,TLI->getScopeLevel(),
TLI->getLineNumber());
                                while (TLI->getTokenNodeType() != DEF) {
                                        KFGUseC hUse1(TLI->getName(),TLI->
getScopeLevel());
                                        P_USEI =
find(hN1.P_Uses.begin(),hN1.P_Uses.end(),hUse1);
                                        if (P_USEI == hN1.P_Uses.end()) {
                                                hN1.P_Uses.push_back(hUse1);
                                                Anzahl_P_Uses++;
                                        }
                                        //hN1.P_Uses.push_back(KFGUseC(TLI->
getName(),TLI->getScopeLevel()));
                                        TLI--;
                                }
                                push_front(hN1);

                                TLI--;
                                KFGListNodeC hN2("NORMAL",NO,(TLI->getScopeLevel()-
1),TLI->getLineNumber());
                                hN2.Defs.push_back(KFGDefC(TLI->getName(),(TLI-
>getScopeLevel()-1)));
                                Anzahl_Defs++;
                                push_front(hN2);
                                TLI--;
                        }
                        break;
                case RET:
                        //Only a temporary implementation; still needs to be
extended
                        //for the case in which an output follows RETURN.
                        push_front(KFGListNodeC("RETURN",RETURN,TLI->
getScopeLevel()));
                        TLI--;
                        break;
                case NL:
                        TLI--;
                        //Produce node which contains DEFs and Uses;
                        if (TLI->getTokenNodeType() == N) {
                                KFGListNodeC helpNode1("NORMAL",NO,TLI->
getScopeLevel(),TLI->getLineNumber());
                                TLI--;
                                if (TLI->getTokenNodeType() == NL) {
                                        TLI++;
                                        helpNode1.Defs.push_back(KFGDefC(TLI-
>getName(),TLI->getScopeLevel()));
                                        Anzahl_Defs++;
                                        helpNode1.Uses.push_back(KFGUseC(TLI-
>getName(),TLI->getScopeLevel()));
                                        Anzahl_Uses++;
                                        TLI--;
                                }
                                else {
                                        TLI++;
                                        while (TLI->getTokenNodeType() != DEF) {
                                                KFGUseC hUse1(TLI->getName(),TLI-
>getScopeLevel());
                                                USEI =
find(helpNode1.Uses.begin(),helpNode1.Uses.end(),hUse1);
                                                if (USEI == helpNode1.Uses.end()) {
                                                        helpNode1.Uses.push_back(hUse);
                                                        Anzahl_Uses++;
                                                }
```

```
                                                    //helpNode1.Uses.push_back
(KFGUseC(TLI->getName(),TLI->getScopeLevel()));
                                                    TLI--;
                                                }
                                                TLI--;
```

```
                                        helpNode1.Defs.push_back(KFGDefC(TLI->
getName(),TLI->getScopeLevel()));
                                        TLI--;
                                        Anzahl_Defs++;
                                        if (TLI->getTokenNodeType() == N) {
                                                helpNode1.Defs.pop_front();
                                                Anzahl_Defs--;                  //in
this is an array and
                        //hence the last node needs to be deleted again.
                                        TLI++;
                //For this, the deleted node needs to be entered into the Use list.
                                        KFGUseC hUse2(TLI->getName(),TLI->
getScopeLevel());
                                        USEI =
find(helpNode1.Uses.begin(),helpNode1.Uses.end(),hUse2);
                                        if (USEI == helpNode1.Uses.end()) {
                                                helpNode1.Uses.push_back(hUse2);
                                                Anzahl_Uses++;
                                        }
                                        TLI--;
                                        helpNode1.Defs.push_back(KFGDefC(TLI-
>getName(),TLI->getScopeLevel()));
                                        helpNode1.Uses.push_back(KFGUseC(TLI-
>getName(),TLI->getScopeLevel()));
                                        Anzahl_Defs++;
                                        Anzahl_Uses++;
                                        TLI--;
                                }
                        }
                        push_front(helpNode1);
                }
                //produce node which contains only Defs;
                else if (TLI->getTokenNodeType() == DEF) {
                        TLI--;
                        KFGListNodeC helpNode1("NORMAL",NO,TLI->
getScopeLevel(),TLI->getLineNumber());
                        helpNode1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                        push_front(helpNode1);
                        Anzahl_Defs++;
                        TLI--;
                }
                else if (TLI->getTokenNodeType() == PF) {
                        TLI--;
                        KFGListNodeC hN1("NORMAL",NO,TLI->getScopeLevel(),
TLI->getLineNumber());
                        hN1.Defs.push_back(KFGDefC(TLI->getName(),TLI-
>getScopeLevel()));
                        hN1.Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                        push_front(hN1);
                        Anzahl_Defs++;
                        Anzahl_Uses++;
                }
                //produce node which contains output variables;
                else {
                        TLI--;
                        KFGListNodeC helpNode1("OUTPUT",OP,TLI->
getScopeLevel(),TLI->getLineNumber());
                        helpNode1.Uses.push_back(KFGUseC(TLI->getName(),TLI-
>getScopeLevel()));
                        push_front(helpNode1);
                        Anzahl_Uses++;
                        TLI--;
                }
                Anweisungen++;
                break;
        default:
                TLI--;
        }
    }
    KnotenNummern();
    KFGListC::iterator KFG = begin();
    KnotenIdentifizierer(KFG);
    addLineInToList();
```

```
        Anzahl_Deklarationen = zaehleDeklarationen(L1);
}
```

```
void KFGListC::KnotenNummern() {
        int KnotenNr = 1;
        KFGListC::iterator KFG1 = begin();
        while (KFG1 != end()) {
                if ((KFG1->getKFGNodeType() == EL) || (KFG1->getKFGNodeType() ==
ETh) ||  (KFG1->getKFGNodeType() == BEl) || (KFG1->getKFGNodeType() == EEl)) {
                        KFG1++;
                }
                else {
                        KFG1->setKFGKnotenNummer(KnotenNr);
                        KnotenNr++;
                        KFG1++;
                }
        }
}


void KFGListC::KnotenIdentifizierer(KFGListC::iterator KFG) {
        int LOOPLevel = 0;
        int THENLevel = 0;
        int ELSELevel = 0;
        KFGListC::iterator KFG1 = KFG;
        KFGListC::iterator LOOPIT = KFG;
        KFGListC::iterator IFIT = KFG;
        KFGListC::iterator ELSEIT = KFG;
        while (KFG1 != end()) {
                switch(KFG1->getKFGNodeType()) {
                case LC:
                        {
                                LOOPLevel = KFG1->getLevel();
                                KFG1++;
                                LOOPIT = KFG1;
                                while ((KFG1->getKFGNodeType() != EL) || (KFG1-
>getLevel() != LOOPLevel)) {
                                        KFG1->setKnotenIdent(LOOP);
                                        KFG1++;
                                }
                                KnotenIdentifizierer(LOOPIT);
                        }
                        break;
                case IFC:
                        {
                                THENLevel = KFG1->getLevel();
                                KFG1++;
                                IFIT = KFG1;
                                while ((KFG1->getKFGNodeType() != ETh) || (KFG1->
getLevel() != THENLevel)) {
                                        KFG1->setKnotenIdent(THEN);
                                        KFG1++;
                                }
                                KnotenIdentifizierer(IFIT);
                        }
                        break;
                case BEl:
                        {
                                ELSELevel = KFG1->getLevel();
                                KFG1++;
                                ELSEIT = KFG1;
                                while ((KFG1->getKFGNodeType() != EEl) || (KFG1-
>getLevel() != ELSELevel)) {
                                        KFG1->setKnotenIdent(ELSE);
                                        KFG1++;
                                }
                                KnotenIdentifizierer(ELSEIT);
                        }
                        break;
                default:
                        //cout << "Ouch" << endl;
                        KFG1++;
                }
                //KFG1++;
        }
}
```

```
void KFGListC::addLineInToList() {
        int zahl;
        char line[256];
        KFGProgListC LP;
        ifstream datei("CodeLine.dat");
        if (!datei) {
                cout <<  "FAULT: Cannot open file 'CodeLine.dat'." << endl;
        }
        else {
                while (!datei.eof()) {
                        datei.getline(line, 255,'\n');
                        for (int i=0; i<strlen(line); i++) {
                                if (line[i] == '"') {
                                        line[i] = '\'';
                                }
                                if (line[i] == '{') {
                                        line[i] = ';';
                                }
                        }
                        zahl = atoi(line);
                        KFGLineNodeC hknoten(line,zahl);
                        LP.push_back(hknoten);
                        //cout << line << " " << zahl << endl;
                }
        }
        addLineToKFG(LP);
}

void KFGListC::addLineToKFG(KFGProgListC & LP) {
        int dummy = 0;
        char help[256];
        KFGListC::iterator KFG1 = end();
        KFGProgListC::iterator PROG1 = LP.end();
        //KFG1--;
        while (KFG1 != begin()) {
                KFG1--;
                if ((KFG1->getKFGNodeType() == EL) || (KFG1->getKFGNodeType() ==
ETh) ||  (KFG1->getKFGNodeType() == BEl) || (KFG1->getKFGNodeType() == EEl)) {
                        KFG1--;
                }
                while ((PROG1 != LP.begin()) && (dummy != 1)) {
                        if (KFG1->getLineNr() == PROG1->getLineNumber()) {
                                strcpy(help,PROG1->getName());
                                KFG1->setCodeLine(help);
                                PROG1--;
                                dummy = 1;
                        }
                        else{
                                PROG1--;
                        }
                }
                PROG1 = LP.end();
                dummy = 0;
        }
}

int KFGListC::zaehleDeklarationen(KFGTokenListC & TL) {
        int zaehler = 0;
        KFGTokenListC::iterator TListI = TL.end();
        while (strcmp(TListI->getName(),"main") != 0) {
                if (TListI->getTokenNodeType() == TD) {
                        zaehler++;
                }
                TListI--;
        }
        return zaehler;
}

//Function for outputting the KFGList to the screen and to a file "CFGList".
void KFGListC::ListeAusgeben() {
        ofstream Ziel1("CFGList.cfg");
        ostream_iterator<KFGListNodeC> Pos(Ziel1, "\n");
        ostream_iterator<KFGDefC> PosD(Ziel1, "\n");
        ostream_iterator<KFGUseC> PosU(Ziel1, "\n");
        ostream_iterator<KFGUseC> PosP(Ziel1, "\n");
```

```
        KFGListC::iterator KLI = begin();
        KFGDefListC::iterator DEF;
        KFGUseListC::iterator USE;
        KFGP_UseListC::iterator P_USE;
        Ziel1 << "START" << endl << endl;
        while (KLI != end()) {
                Ziel1 << "Line  " << KLI->getLineNr() << endl;
                Ziel1 << "Zeile " << KLI->getCodeLine() << endl;
                Ziel1 << "Knoten " << KLI->getKnotenNummer() << endl;
                Ziel1 << "KnotenTyp " << KLI->getKnotenIdent() << endl;
                Ziel1 << KLI->getStatement() << "     " << KLI->getLevel() << endl;
                //*Pos++ = *KLI;
                DEF = KLI->Defs.begin();
                USE = KLI->Uses.begin();
                P_USE = KLI->P_Uses.begin();
                while (DEF != KLI->Defs.end()) {
                        //*PosD++ = *DEF;
                        Ziel1 << "DEF:         " << DEF->getDef() << "        " <<
DEF->getScopeLevelD() << endl;
                        DEF++;
                }
                while (USE != KLI->Uses.end()) {
                        //*PosU++ = *USE;
                        Ziel1 << "USE:         " << USE->getUse() << "        " <<
USE->getScopeLevelU() << endl;
                        USE++;
                }
                while (P_USE != KLI->P_Uses.end()) {
                        //*PosP++ = *P_USE;
                        Ziel1 << "P_USE:              " << P_USE->getUse() << "
        " << P_USE->getScopeLevelU() << endl;
                        P_USE++;
                }
                Ziel1 << endl;
                KLI++;
        }
        Ziel1 << "STOP" << endl << endl;
        Ziel1 << "BASIC VARIABLES:" << endl;
        Ziel1 << "Number of empty branches: " << "0" << endl;
        Ziel1 << "Number of decisions: " << Entscheidungen << endl;
        Ziel1 << "Number of instructions: " << Anweisungen << endl;
        Ziel1 << "Number of atomic predicates: " << Entscheidungen << endl;
        Ziel1 << "Number of predicates: " << Entscheidungen << endl;
        Ziel1 << "Number of atomic predicates which are arithm. relations: " << "0"
<< endl;
        Ziel1 << "Number of loop decisions: " << Schleifenentscheidungen << endl;
        Ziel1 << "Number of non-interleaved loop deicisions: " << "0" << endl;
        Ziel1 << "Number of count loop decisions: " << Zaehlschleifenentscheidungen
<< endl;
        Ziel1 << "Number of declarations of structured data types: " << "0" <<
endl;
        Ziel1 << "Number of declarations: " << Anzahl_Deklarationen << endl;
        Ziel1 << "Number of real element declarations: " << "0" << endl;
        Ziel1 << "Number of basic type declarations: " << Anzahl_Deklarationen <<
endl;
        Ziel1 << "Number of Defs:     " << Anzahl_Defs << endl;
        Ziel1 << "Number of Uses:     " << Anzahl_Uses << endl;
        Ziel1 << "Number of P-uses:   " << Anzahl_P_Uses << endl << endl;
        basisgroessenInDatei();
}

void KFGListC::basisgroessenInDatei() {
        ofstream datei("Basisgroessen.bgd", ios::out);
        datei << "0" << endl;
        datei << Entscheidungen << endl;
        datei << Anweisungen << endl;
        datei << Entscheidungen << endl;
        datei << Entscheidungen << endl;
        datei << "0" << endl;
        datei << Schleifenentscheidungen << endl;
        datei << "0" << endl;
        datei << Zaehlschleifenentscheidungen << endl;
        datei << "0" << endl;
        datei << Anzahl_Deklarationen << endl;
        datei << "0" << endl;
        datei << Anzahl_Deklarationen << endl;
```

```
        datei << "0" << endl;
        datei << Anzahl_Defs << endl;
        datei << Anzahl_Uses << endl;
        datei << Anzahl_P_Uses << endl;
```

5

```
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
        datei << "0" << endl;
}
```

```
#include "KFGListe.h"

KFGListeT KFGListe;

void KFGListeAusgeben(KFGListeT & L) {

        KFGListeT::iterator I = L.begin();

                while(I != L.end())
                        cout << *I++ << ' ';
                cout << " size() = " <<  L.size() << endl;
}

#include "KFGListNode.h"

#include <iostream>

        ostream& operator << ( ostream& os, const KFGListNodeC& Node){

                os << endl << "KnotenNr:" << Node.KnotenNr << " Typ:" <<
Node.KnotenIdent << endl << Node.Statement << " Level:" << Node.Level;
                return os;
        }

        int KFGListNodeC::DummyNodeNr=1;


#include "KFGNode.h"

#include <iostream>


        ostream& operator << ( ostream& os, const KFGNodeC& Node){

                os << Node.Name << " " << Node.ScopeLevel;
                return os;
        }

#include "KFGTokenList.h"
```

```
void KFGTokenListC::KFGListeAusgeben() {

                ofstream Ziel("CFGTokenList");

                ostream_iterator<KFGTokenNodeC> oPos(Ziel, "\n");

                KFGTokenListC::iterator I = begin();

                while(I != end()) {
                        //*oPos++ = I->getName();
                        *oPos++ = *I;
                        //cout << I->getName() << " " << I->getScopeLevel() << " "
<< I->getTokenNodeType() << endl;
                        I++;
                }
                cout << " size() = " <<  size() << endl;
        }




#include "KFGTokenNode.h"
#include <iostream>

        ostream& operator << ( ostream& os, const KFGTokenNodeC& Node){
                os << Node.Name << " " << Node.ScopeLevel <<  " " <<
Node.TokenNodeType << " " << Node.LineNumber << endl;
                return os;
        }


#include "KFGUse.h"

#include <iostream>

        ostream& operator << ( ostream& os, const KFGUseC& Node){
                os << "USE:            " << Node.Use << "     " << Node.ScopeLevelU;
                return os;
        }


/*

 * Main program to test C++ grammar and preprocessor

 */


#include "JLStr.h"

#include "tokens.h"

#include "DLGLexer.h"
#include "BufferedCPreParser.h"
#include "CPreToCPPBuffer.h"
#include "JLTokenBuffer.h"
#include "CPPParserSym.h"
#include "KFGTokenList.h"
#include "KFGList.h"
#include "graph.h"
#include "AS_Slice.h"
#include "uwggraph.h"
#include "KFGProgList.h"
#include <iostream>

static void usage(char* progname);

int main(int argc, char *argv[])
{
   // For reporting memory usage
   char *p = new char[100000];
   long heap1 = (long)(void*)p;
```

```
   delete [] p;

   // Parse command-line options
   JLStr includePath;
   JLStr definitions;
   FILE *inputFile = stdin;
   bool doTraceParse = false;
   bool doTraceInclude = false;
   bool doDump = false;
   char *progname = *argv;

   argc--;
   argv++;

   ofstream datei("CodeLine.dat", ios::trunc);

   while (argc > 0)
   {
      if (argv[0][0] == '-')
      {
         switch (argv[0][1])
         {
            case 'I':
               if (argv[0][2] != 0)
               {
                  // argument is part of "-I"
                  includePath += ';';
                  includePath += &argv[0][2];
               } else if (argc > 1)
               {
                  argc--;
                  argv++;
                  includePath += ';';
                  includePath = *argv;
               } else {
                  usage(progname);
               }
               break;
            case 'd':
               if (strcmp(argv[0], "-dump") == 0)
               {
                  doDump = true;
               }
               break;
            case 'D':
               if (argv[0][2] != 0)
               {
                  // argument is part of "-D"
                  definitions += ';';
                  definitions += &argv[0][2];
               } else if (argc > 1)
               {
                  // argument follows "-D"
                  argc--;
                  argv++;
                  definitions += ';';
                  definitions = *argv;
               } else {
                  usage(progname);
               }
               break;
            case 't':
               if (strcmp(*argv, "-traceParse") == 0)
               {
                  doTraceParse = true;
               } else if (strcmp(*argv, "-traceInclude") == 0)
               {
                  doTraceInclude = true;
               } else {
                  usage(progname);
               }
               break;
            default:
               usage(progname);
               break;
```

```
    }
  } else {
    // must be the input filename
```

```
            if (inputFile != stdin)
            {
               // already have one
               usage(progname);
            }
            FILE *fp = fopen(*argv, "r");
            if (fp == NULL)
            {
               fprintf(stderr, "%s: cannot open %s for input\n", progname, *argv);
               exit(0);
            }
            inputFile = fp;
         }
         argc--;
         argv++;
      }

      //printf("%s\n%s\n%s\n",includePath,definitions,inputFile);

      // Create input stream, lexer
      DLGFileInput input(inputFile);
      DLGLexer scanner(&input);
      FastToken tok;
      scanner.setToken(&tok);

      // Create preprocessor parser.  Note that the preprocessor parser has
      // a built-in token buffer in the form of an input stack.
           BufferedCPreParser preprocessor(&scanner);
           preprocessor.init();

      // set include path and defines directly for debugging
      if (includePath.length() == 0)
      {
         includePath = "c:\\devstudio\\vc\\include;\\msdev\\devstudio\\vc\\mfc
\\include";
      }
      if (definitions.length() == 0)
      {
         definitions =
            "__cplusplus;"
            "__DATE__=\"date\";"
            "__FILE__=\"filename\";"
            "__LINE__=1;"
            "__TIME__=\"time\";"
            "__TIMESTAMP__=\"timestamp\";"
            "_M_IX86=400;"
            "_MSC_VER=1100;"
            "_WIN32;"
            "_INTEGRAL_MAX_BITS=64";
      }

      // Set preprocessor options
      preprocessor.SetIncludePath(includePath);
      preprocessor.SetDefinitions(definitions);

      // Set options in parser to make it act standard with MS Extensions
      preprocessor.SetOption(CPreParserImp::OptMSExtensions, true);
      preprocessor.SetOption(CPreParserImp::OptBool, true);
      preprocessor.SetOption(CPreParserImp::OptWCharT, true);
      preprocessor.SetOption(CPreParserImp::OptPragmas, true);
      preprocessor.SetOption(CPreParserImp::OptExpandPragmas, true);
      preprocessor.SetOption(CPreParserImp::OptTrackInclude, doTraceInclude);

      // Buffer to store tokens output by preprocessor
      CPreToCPPBuffer pipe2(&preprocessor);

      // Connect preprocessor to second token buffer so that the
      // preprocessor can unilaterally squirt new tokens into the buffer
      preprocessor.SetBuffer(&pipe2);

      // Token buffer for the usual backtracking, etc.
      JLTokenBuffer pipe3(&pipe2, CPPParserSym::ConstLLK);

      // Create the C++ parser
      CPPParserSym parser(&pipe3);
```

```
// Set parser options to look like MSVC++
parser.SetOption(CPPParserSym::OptPragmaMSPacking, true);
parser.SetOption(CPPParserSym::OptMSTypedefHack, true);
```

```
      //preprocessor.doTrace(doTraceParse);
   parser.doTrace(doTraceParse);

   // Process top-level rule
   parser.init();

   KFGTokenListC L1;
   parser.translation_unit(L1);

   // Close the input file stream
   if (inputFile != stdin)
   {
      fclose(inputFile);
   }

   // For reporting memory usage
   // For reporting memory usage
   p = new char[100000];
   long heap2 = (long)(void*)p;
   delete [] p;
   cout << "Approx heap usage before dump: " << (heap2 - heap1) << endl;

   if (doDump)
   {
      // Dump the scope hierarchy
      cout << "Dump of scope hierarchy" << endl;
      parser.DumpScopes();
      cout << endl;
   }

   // For reporting memory usage
   // For reporting memory usage
   p = new char[100000];
   long heap3 = (long)(void*)p;
   delete [] p;
   cout << "Approx heap usage after dump: " << (heap3 - heap1) << endl;
   L1.KFGListeAusgeben();
   //L1.KFGTokenListToKFGList();

   KFGListC L2;
   //KFGListC::iterator Iter;
   L2.TokenList2KFGList(L1);
   L2.ListeAusgeben();

   SliceC Slice;
   Slice.startBuildSlice(L2);
   //Slice.buildTestGraph(L2);
   //cout << Slice;

   uwggraphC uwggraph;
   //uwggraph.buildIfTree(Slice);
   uwggraph.startBuildFT(Slice);
   cout << uwggraph;

   return 0;
}

static void usage(char* progname)
{
   fprintf(
      stderr,
      "usage: %s [-I directories] [-D definitions] [-traceParse] [ -traceInclude ]
[-o ofile] [ifile]\n"
      "    -I directories: semicolon-separated list of include directories\n"
      "    -D definitions: semicolon-separated list of definitions, like:\n"
      "                    symbol1;symbol2=value2\n"
      "    -traceParse: output debugging information\n"
      "    -traceInclude: output trace of include stack\n"
      "    -dump: dump type and scope hierarchy\n"
      "    ifile: name of input file (otherwise it uses stdin)\n",
      progname
   );
   exit(1);
}

#include <iostream>
```

```
#include <iterator>
#include <algorithm>
#include <fstream>
#include "uwggraph.h"

using namespace std;

SliceC::iterator uwggraphC::findOutputNode(SliceC & S1) {
        int dummy = 0;
        SliceC::iterator SL1 = S1.begin();
        SliceC::iterator HELP = S1.begin();
        while ((SL1 != S1.end()) && (dummy != 1)) {
                if ( (*SL1).first.getKFGNodeType() == OP ) {
                        HELP = SL1;
                        dummy = 1;
                }
                SL1 ++;
        }
        return HELP;
}


void uwggraphC::startBuildFT(SliceC & S1) {
        int pos1 = 0;
        int Posi1 = 0;
        int SliceNr = 0;
        int LineNr = 0;
        int inLoop = 0;
        SliceAusgeben(S1);
        char NodeText[128];
        char Bemerkung[1000];
        SliceC::iterator SL1 = findOutputNode(S1);
        SliceNr = (*SL1).first.getKnotenNummer();
        LineNr = (*SL1).first.getLineNr();
        sprintf(NodeText,"Op%d",SliceNr);
        strcpy(Bemerkung,SL1->first.getCodeLine());
        //sprintf(Bemerkung,"Line %d",LineNr);
        SliceC::Nachfolger::iterator BEGIN = (*SL1).second.begin();
        SliceC::Nachfolger::iterator END = (*SL1).second.end();
        while (BEGIN != END) {
                if (BEGIN->second == KFK) {
                        uwgknotenC hknoten1(EFFECT,SliceNr,NodeText,Bemerkung);
                        Posi1 = insert(hknoten1);
                        SliceC::iterator SLPUSE = findLastPUse(S1, SliceNr);
                        checkLoopOrCond(S1, SLPUSE, SL1, Posi1);
                        inLoop = 1;
                        BEGIN++;
                }
                else {
                        BEGIN++;
                }
        }
        if (inLoop == 0) {
                uwgknotenC hknoten1(EFFECT,SliceNr,NodeText,Bemerkung);
                uwgknotenC hknoten2(OR,"OR");
                insert(hknoten1,hknoten2,0);
                pos1 = size()-1;
                uwgknotenC hknoten3(CAUSE,SliceNr,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                addFirstNodesToFT(S1, SliceNr, pos1);
        }
        check();
}

void uwggraphC::checkLoopOrCond(SliceC & S1, SliceC::iterator SLPUSE,
SliceC::iterator SLORIG, int Pos1) {
        int PosRet = 0;
        switch (SLPUSE->first.getKFGNodeType()) {
        case IFC:
                PosRet = buildInIfTree(S1, SLPUSE, SLORIG, Pos1);
                //cout << "Test1a: " << SLHELP->first.getKnotenNummer() << endl;
                break;
        case LC:
                buildInLoopTree(S1, Pos1, SLPUSE, SLORIG);
                //cout << "Test1b: " << SLHELP->first.getKnotenNummer() << endl;
```

```
                break;
        default:
                cout << "Ups" << endl;
        }
}

void uwggraphC::addFirstNodesToFT(SliceC & S1, int KnotenNr, int Pos1) {
        int helpNr = 0;
        int ret = 0;
        int pos = 0;
        int KnotenNr1 = 0;
        int LineNr1 = 0;
        char NodeText[128];
        char Bemerkung[1000];
        SliceC::iterator NODE = defineIterator(S1, KnotenNr);
        SliceC::Nachfolger::iterator BEGIN = NODE->second.begin();
        SliceC::Nachfolger::iterator END = NODE->second.end();
        while (BEGIN != END) {
                helpNr = BEGIN->first;
                KnotenNr1 = S1[helpNr].first.getKnotenNummer();
                SliceC::iterator NODE1 = defineIterator(S1,KnotenNr1);
                if (NODE1->first.getLevel() > 1) {
                        SliceC::iterator PUSE = findLastPUse(S1, KnotenNr1);
                        switch (PUSE->first.getKFGNodeType()) {
                        case IFC:
                                ret = buildInIfTree(S1,PUSE,NODE1,Pos1);
                                break;
                        case LC:
                                buildInLoopTree(S1,Pos1,PUSE,NODE1);
                                break;
                        default:
                                cout << "Something is wrong!" << endl;
                        }
                }
                else {
                        LineNr1 = S1[helpNr].first.getLineNr();
                        sprintf(NodeText,"Op%d",KnotenNr1);
                        //sprintf(Bemerkung,"Line %d",LineNr1);
                        strcpy(Bemerkung,S1[helpNr].first.getCodeLine());
                        uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText,Bemerkung);
                        pos = insert(hknoten1);
                        verbindeEcken(Pos1,pos,0);
                        addFirstNodesToFT(S1, KnotenNr1, Pos1);
                }
                BEGIN++;
        }
}

SliceC::iterator uwggraphC::findLastPUse(SliceC & S1, int SliceNr) {
        int helpNr1 = 0;
        int SliceNr1 = 0;
        SliceC::iterator SL1 = defineIterator(S1, SliceNr);
        SliceC::iterator SLHELP = SL1;
        SliceC::Nachfolger::iterator BEGIN = (*SL1).second.begin();
        SliceC::Nachfolger::iterator END = (*SL1).second.end();
        while (BEGIN != END) {
                if(BEGIN->second == KFK) {
                        helpNr1 = BEGIN->first;
                        SliceNr1 = S1[helpNr1].first.getKnotenNummer();
                        SLHELP = defineIterator(S1, SliceNr1);
                        SLHELP = findLastPUse(S1, SliceNr1);
                }
                BEGIN++;
        }
        return SLHELP;
}

SliceC::iterator uwggraphC::findLastPUse2(SliceC & S1, int SliceNr, int RefNr) {
        int helpNr1 = 0;
        int SliceNr1 = 0;
        SliceC::iterator SLRETURN = defineIterator(S1, SliceNr);
        SliceC::iterator SLHELP = SLRETURN;
        SliceC::Nachfolger::iterator BEGIN = SLRETURN->second.begin();
        SliceC::Nachfolger::iterator END = SLRETURN->second.end();
        while (BEGIN != END) {
```

```
                if(BEGIN->second == KFK) {
                        helpNr1 = BEGIN->first;
                        SliceNr1 = S1[helpNr1].first.getKnotenNummer();
                        SLHELP = defineIterator(S1, SliceNr1);
                        SLHELP = findLastPUse2(S1, SliceNr1, RefNr);
                        if (SLHELP->first.getKnotenNummer() != RefNr) {
                                SLRETURN = SLHELP;
                        }
                }
                BEGIN++;
        }
        return SLRETURN;
}


SliceC::iterator uwggraphC::lookForNextPUse(SliceC & S1, SliceC::iterator SNODE,
SliceC::iterator SLPUSE) {
        int helpNr1 = 0;
        int helpNr2 = 0;
        int helpNr3 = 0;
        int dummy = 0;
        int dummy1 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int RefNr = SLPUSE->first.getKnotenNummer();
        SliceC::iterator HELP = SNODE;
        SliceC::iterator RETURN = SNODE;
        SliceC::Nachfolger::iterator BEGIN1 = SNODE->second.begin();
        SliceC::Nachfolger::iterator END1 = SNODE->second.end();
        while ((BEGIN1 != END1) && (dummy != 1)){
                helpNr1 = BEGIN1->first;
                SliceC::Nachfolger::iterator BEGIN2 = S1[helpNr1].second.begin();
                SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                while ((BEGIN2 != END2) && (dummy1 != 1)) {
                        if (BEGIN2->second == KFK) {
                                helpNr2 = BEGIN2->first;
                                if (S1[helpNr2].first.getKnotenNummer() != RefNr) {
                                        KnotenNr1 = S1[helpNr2].first.getKnotenNummer();
                                        RETURN = defineIterator(S1, KnotenNr1);
                                        SliceC::Nachfolger::iterator BEGIN3 = RETURN-
>second.begin();
                                        SliceC::Nachfolger::iterator END3 = RETURN-
>second.end();
                                        while (BEGIN3 != END3) {
                                                if (BEGIN3->second == KFK) {
                                                        helpNr3 = BEGIN3->first;
                                                        if
(S1[helpNr3].first.getKnotenNummer() != RefNr) {
                                                                KnotenNr2 = S1[helpNr3]
.first.getKnotenNummer();
                                                                RETURN = findLastPUse2
(S1, KnotenNr2, RefNr);
                                                        }
                                                }
                                                BEGIN3++;
                                        }
                                        dummy1 = 1;
                                        dummy = 1;
                                }
                                else {
                                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer ();
                                    HELP = defineIterator(S1, KnotenNr1);
                                        RETURN = lookForNextPUse(S1, HELP, SLPUSE);
                                }
                        }
                        BEGIN2++;
                }
                BEGIN1++;
        }
        return RETURN;
}

SliceC::iterator uwggraphC::returnCondNode(SliceC & S1, int SliceNr)  {
        int helpNr = 0;
```

```
        int dummy = 0;
        int dummy2 = 0;
        int SliceNr2 = 0;
        SliceC::iterator SL1 = defineIterator(S1, SliceNr);
        SliceC::iterator HELP = SL1;
        SliceC::Nachfolger::iterator BEGIN = (*SL1).second.begin();
        SliceC::Nachfolger::iterator END = (*SL1).second.end();
        while ((BEGIN != END) && (dummy != 1)) {
                helpNr = BEGIN->first;
                SliceNr = S1[helpNr].first.getKnotenNummer();
                HELP = defineIterator(S1, SliceNr);
                SliceC::Nachfolger::iterator BEGIN2 = (*HELP).second.begin();
                SliceC::Nachfolger::iterator END2 = (*HELP).second.end();
                while ((BEGIN2 != END2) && (dummy2 != 1)) {
                        if ((*BEGIN2).second == KFK) {
                                helpNr = (*BEGIN).first;
                                SliceNr2 = S1[helpNr].first.getKnotenNummer();
                                HELP = defineIterator(S1, SliceNr2);
                                dummy2 = 1;
                                dummy = 1;
                        }
                        BEGIN2++;
                }
                BEGIN++;
        }
        return HELP;
}

//Function investigates whether or not two control structures are interleaved.
int uwggraphC::checkPUses1(SliceC & S1, SliceC::iterator SLPUSEREF,
SliceC::iterator SLPUSE) {
        int helpreturn = 0;
        int KnotenNummerRef = 0;
        int helpNr1 = 0;
        int KnotenNr1 = 0;
        KnotenNummerRef = SLPUSE->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN1 = SLPUSEREF->second.begin();
        SliceC::Nachfolger::iterator END1 = SLPUSEREF->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == KFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        if (KnotenNr1 == KnotenNummerRef) {
                                helpreturn = 1;
                        }
                        else {
                                SLPUSEREF = defineIterator(S1,KnotenNr1);
                                helpreturn = checkPUses1(S1,SLPUSEREF,SLPUSE);
                        }
                }
                BEGIN1++;
        }
        return helpreturn;
}

int uwggraphC::checkPUses2(SliceC & S1, SliceC::iterator SLPUSEREF,
SliceC::iterator SLPUSE) {
        int helpreturn = 0;
        int KnotenNummerRef = 0;
        int helpNr1 = 0;
        int KnotenNr1 = 0;
        KnotenNummerRef = SLPUSEREF->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN1 = SLPUSE->second.begin();
        SliceC::Nachfolger::iterator END1 = SLPUSE->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == KFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        if (KnotenNr1 == KnotenNummerRef) {
                                helpreturn = 1;
                        }
                        else {
                                SLPUSE = defineIterator(S1,KnotenNr1);
                                helpreturn = checkPUses2(S1,SLPUSEREF,SLPUSE);
                        }
```

```
                }
                BEGIN1++;
        }
        return helpreturn;
}

int uwggraphC::buildInIfTree(SliceC & S1, SliceC::iterator SLIF, SliceC::iterator
SLORIG, int pos1) {
        int pos2 = 0;
        int posIFOR = 0;
        int posIFDFOR = 0;
        int posIFKFOR = 0;
        int posIFDFORA2 = 0;
        char NodeText[128];
        char Bemerkung[1000];
        int KnotenNr = SLIF->first.getKnotenNummer();
        int LineNr = SLIF->first.getLineNr();
        sprintf(Bemerkung,"Verzweigung(%d)",LineNr);
        strcpy(NodeText,SLIF->first.getCodeLine());
        uwgknotenC hknoten1(OR,NodeText,Bemerkung);
        pos2 = insert(hknoten1);
        posIFOR = size() - 1;
        verbindeEcken(pos1, pos2,0);
        sprintf(NodeText,"Verzweigung_KF(%d)",LineNr);
        uwgknotenC hknoten5(OR,NodeText);
        posIFKFOR = insert(hknoten5);
        verbindeEcken(pos2,posIFKFOR,0);
        build_IFKF_Part(S1, posIFKFOR, SLIF);
        switch (SLORIG->first.getKnotenIdent()) {
        case THEN:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.1(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.1(%d)",LineNr);
                sprintf(NodeText,"Alt.1(%d)",LineNr);
                uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.1(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFOR = size() - 1;
                build_A1_Part(S1, posIFDFOR, SLIF, SLORIG);
                }
                break;
        case ELSE:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.2(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.2(%d)",LineNr);
                sprintf(NodeText,"Alt.2(%d)",LineNr);
                uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.2(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFORA2 = size() - 1;
                build_A1_Part(S1, posIFDFORA2, SLIF, SLORIG);
                }
                break;
        default:
                cout << "Ouch!!!" << endl;
        }
        return posIFDFOR;
}

void uwggraphC::build_A1_Part(SliceC & S1, int posIFDFOR, SliceC::iterator SLIF,
SliceC::iterator SLNODE) {
        int pos0 = 0;
        int pos1 = 0;
        int ret = 0;
        int dummy = 0;
        int checkNr = 1;
```

```
        int checkPUse = 1;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int KnotenNr0 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int LineNr0 = 0;
        int LineNr1 = 0;
        char NodeText0[128];
        char Bemerkung0[1000];
        KnotenNr0 = SLNODE->first.getKnotenNummer();
        LineNr0 = SLNODE->first.getLineNr();
        sprintf(NodeText0,"Op%d",KnotenNr0);
        //sprintf(Bemerkung0,"Line %d",LineNr0);
        strcpy(Bemerkung0,SLNODE->first.getCodeLine());
        uwgknotenC hknoten0(CAUSE,KnotenNr0,NodeText0,Bemerkung0);
        pos0 = insert(hknoten0);
        verbindeEcken(posIFDFOR,pos0,0);
        SliceC::Nachfolger::iterator BEGIN0 = SLNODE->second.begin();
        SliceC::Nachfolger::iterator END0 = SLNODE->second.end();
        while (BEGIN0 != END0) {
                if (BEGIN0->second == DFK) {
                        helpNr1 = BEGIN0->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        SLNODE = defineIterator(S1,KnotenNr1);
                        if (KnotenNr1 < KnotenNr0) {
                                SliceC::Nachfolger::iterator BEGIN1 = SLNODE-
>second.begin();
                                SliceC::Nachfolger::iterator END1 = SLNODE-
>second.end();
                                while (BEGIN1 != END1) {
                                        if (BEGIN1->second == KFK) {
                                                helpNr2 = BEGIN1->first;
                                                KnotenNr2 =
S1[helpNr2].first.getKnotenNummer();
                                                SliceC::iterator HELPPUSE =
defineIterator(S1,KnotenNr2);
                                                checkNr = checkUWGNode(KnotenNr2);
                                                if (SLIF->first.getKnotenNummer() >
KnotenNr2) {
                                                        checkPUse =
checkPUses1(S1,SLIF,HELPPUSE);
                                                }
                                                else {
                                                        checkPUse =
checkPUses2(S1,SLIF,HELPPUSE);
                                                }
                                                if (KnotenNr2 != SLIF-
>first.getKnotenNummer()) {
                                                        if (checkPUse == 0) {
                                                                dummy = 1;
                                                                HELPPUSE = findOutmost
PUse(S1,HELPPUSE);
                                                                switch (HELPPUSE->
first.getKFGNodeType()) {
                                                                case LC:
                                                                        buildInLoopTree
(S1, posIFDFOR, HELPPUSE, SLNODE);
                                                                        break;
                                                                case IFC:
                                                                        ret =
buildInIfTree(S1, HELPPUSE, SLNODE, posIFDFOR);
                                                                        break;
                                                                default:
                                                                        cout <<
"Incorrect path in A1! " << endl;
                                                                }
                                                        }
                                                        else {
                                                                if (checkNr == 0) {
                                                                        dummy = 1;
                                                                        HELPPUSE =
findOutmostPUse(S1,HELPPUSE);
                                                                        switch
(HELPPUSE->first.getKFGNodeType()) {
```

```
                                                        case LC:

        buildInLoopTree(S1, posIFDFOR, HELPPUSE, SLNODE);
                                                               break;
5                                                       case IFC:
```

```
                                                                    ret =
buildInIfTree(S1, HELPPUSE, SLNODE, posIFDFOR);
                                                                    break;
                                                            default:
                                                                    cout <<
"Incorrect path! " << endl;
                                                                    }
                                                            }
                                                    }
                                            }
                                    }
                                    BEGIN1++;
                            }
                            if (dummy == 0) {
                                    LineNr1 = S1[helpNr1].first.getLineNr();
                                    sprintf(NodeText0,"Op%d",KnotenNr1);
                                    //sprintf(Bemerkung0,"Line %d",LineNr1);

        strcpy(Bemerkung0,S1[helpNr1].first.getCodeLine());
                                    uwgknotenC
hknoten1(CAUSE,KnotenNr1,NodeText0,Bemerkung0);
                                    pos1 = insert(hknoten1);
                                    verbindeEcken(posIFDFOR,pos1,0);
                                    build_A1_Part(S1, posIFDFOR, SLIF, SLNODE);
                            }
                    }
            }
            BEGIN0++;
            dummy = 0;
    }
}


void uwggraphC::build_IFKF_Part(SliceC & S1, int posKFOR, SliceC::iterator SLPUSE)
{
    int pos1 = 0;
    int ret = 0;
    int dummy = 0;
    int checkNr = 1;
    int checkPUse = 1;
    int helpNr1 = 0;
    int helpNr2 = 0;
    int KnotenNr0 = 0;
    int KnotenNr1 = 0;
    int KnotenNr2 = 0;
    int LineNr0 = 0;
    int LineNr1 = 0;
    char NodeText[128];
    char Bemerkung[1000];
    KnotenNr0 = SLPUSE->first.getKnotenNummer();
    LineNr0 = SLPUSE->first.getLineNr();
    sprintf(NodeText,"Op%d",KnotenNr0);
    //sprintf(Bemerkung,"Line %d",LineNr0);
    strcpy(Bemerkung,SLPUSE->first.getCodeLine());
    uwgknotenC hknoten1(CAUSE,KnotenNr0,NodeText,Bemerkung);
    pos1 = insert(hknoten1);
    verbindeEcken(posKFOR,pos1,0);
    SliceC::Nachfolger::iterator BEGIN0 = SLPUSE->second.begin();
    SliceC::Nachfolger::iterator END0 = SLPUSE->second.end();
    while (BEGIN0 != END0) {
        if (BEGIN0->second == DFK) {
            helpNr1 = BEGIN0->first;
            KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
            SliceC::iterator SLNODE = defineIterator(S1,KnotenNr1);
            if (KnotenNr1 < KnotenNr0) {
                SliceC::Nachfolger::iterator BEGIN1 = SLNODE-
>second.begin();
                SliceC::Nachfolger::iterator END1 = SLNODE-
>second.end();
                while (BEGIN1 != END1) {
                    if (BEGIN1->second == KFK) {
                        helpNr2 = BEGIN1->first;
                        KnotenNr2 =
S1[helpNr2].first.getKnotenNummer();
                        SliceC::iterator HELPPUSE =
defineIterator(S1,KnotenNr2);
```

```
KnotenNr2) {

checkPUses1(S1,SLPUSE,HELPPUSE);
```

```
checkNr = checkUWGNode(KnotenNr2);
if (SLPUSE->first.getKnotenNummer() >

        checkPUse =

}
```

```
                                                else {
                                                        checkPUse =
    checkPUses2(S1,SLPUSE,HELPPUSE);
                                                }
 5                                              if (KnotenNr2 != SLPUSE-
    >first.getKnotenNummer()) {
                                                        if (checkPUse == 0) {
                                                                dummy = 1;
                                                                HELPPUSE =
10  findOutmostPUse(S1,HELPPUSE);
                                                                switch (HELPPUSE-
    >first.getKFGNodeType()) {
                                                                case LC:
15          buildInLoopTree(S1, posKFOR, HELPPUSE, SLNODE);
                                                                        break;
                                                                case IFC:
                                                                        ret =
    buildInIfTree(S1, HELPPUSE, SLNODE, posKFOR);
20                                                                      break;
                                                                default:
                                                                        cout <<
    "Incorrect path in A1! " << endl;
                                                                }
25                                                      }
                                                        else {
                                                                if (checkNr == 0) {
                                                                        dummy = 1;
                                                                        HELPPUSE =
30  findOutmostPUse(S1,HELPPUSE);
                                                                        switch
    (HELPPUSE->first.getKFGNodeType()) {
                                                                        case LC:
          buildInLoopTree(S1, posKFOR, HELPPUSE, SLNODE);
35                                                                              break;
                                                                        case IFC:
                                                                                ret =
    buildInIfTree(S1, HELPPUSE, SLNODE, posKFOR);
                                                                                break;
40                                                                      default:
                                                                                cout <<
    "Incorrect path! " << endl;
                                                                        }
                                                                }
45                                                      }
                                                }
                                        }
                                        BEGIN1++;
                                }
50                              if (dummy == 0) {
                                        LineNr1 = S1[helpNr1].first.getLineNr();
                                        sprintf(NodeText,"Op%d",KnotenNr1);
                                        //sprintf(Bemerkung,"Line %d",LineNr1);
                                        strcpy(Bemerkung,S1[helpNr1].first .getCodeLine());
55                                      uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText,
    Bemerkung);
                                        pos1 = insert(hknoten1);
                                        verbindeEcken(posKFOR,pos1,0);
                                        build_IFKF_Part(S1, posKFOR, SLNODE);
60                              }
                        }
                }
                BEGIN0++;
                dummy = 0;
65      }
}

void uwggraphC::buildInLoopTree(SliceC &  S1, int Pos1, SliceC::iterator SLPUSE,
SliceC::iterator SLNODE) {
70      int posi1 = 0;
        int pos3 = 0;
        int posLOOP_KF = 0;
        int posOR_D1 = 0;
        int posOR_D2 = 0;
75      int posAND_KF = 0;
        int SliceNr = 0;
        int NodeNrD2 = 0;
```

```
        int KnotenNr = 0;
        int LineNr = 0;
        KnotenNr = SLPUSE->first.getKnotenNummer();
        LineNr = SLPUSE->first.getLineNr();
        char NodeText[128];
        char Bemerkung[1000];
        sprintf(Bemerkung,"Schleife(%d)",KnotenNr);
        strcpy(NodeText,SLPUSE->first.getCodeLine());
        uwgknotenC hknoten1(OR,NodeText,Bemerkung);
        posi1 = insert(hknoten1);
        verbindeEcken(Pos1,posi1,0);
        sprintf(NodeText,"Schleife_DF_Durchl.1(%d)",LineNr);
        uwgknotenC hknoten2(AND,NodeText);
        sprintf(NodeText,"Schleife_DF_Durchl.+(%d)",LineNr);
        uwgknotenC hknoten3(AND,NodeText);
        pos3 = insert(hknoten3);
        verbindeEcken(posi1,pos3,0);
        sprintf(NodeText,"Schleife_KF(%d)",LineNr);
        uwgknotenC hknoten4(OR,NodeText);
        insert(hknoten1,hknoten2,0);
        insert(hknoten1,hknoten4,0);
        posLOOP_KF = size() - 1;
        sprintf(Bemerkung,"Durchl.1(%d)",LineNr);
        sprintf(NodeText,"Durchl.1(%d)",LineNr);
        uwgknotenC hknoten5(CAUSE,NodeText,Bemerkung);
        sprintf(NodeText,"Durchl.1(%d)",LineNr);
        uwgknotenC hknoten6(OR,NodeText);
        insert(hknoten2,hknoten5,0);
        insert(hknoten2,hknoten6,0);
        posOR_D1 = size() - 1;
        sprintf(Bemerkung,"Durchl.+(%d)",LineNr);
        sprintf(NodeText,"Durchl.+(%d)",LineNr);
        uwgknotenC hknoten7(CAUSE,NodeText,Bemerkung);
        sprintf(NodeText,"Durchl.+(%d)",LineNr);
        uwgknotenC hknoten8(OR,NodeText);
        insert(hknoten3,hknoten7,0);
        posOR_D2 = insert(hknoten8);
        verbindeEcken(pos3, posOR_D2,0);
        sprintf(NodeText,"KF(%d)",LineNr);
        uwgknotenC hknoten9(AND,NodeText);
        insert(hknoten4,hknoten9,0);
        insert(hknoten3,hknoten5,0);
        posAND_KF = size() - 1;
        insert(hknoten9,hknoten5,0);
        NodeNrD2 = buildIn_LoopKF_ANDPart(S1, posAND_KF, SLPUSE);
        buildIn_LoopKF_ORPart(S1, posLOOP_KF, SLPUSE, NodeNrD2);
        //buildIn_D2_Part(S1, posOR_D2, SLPUSE, SLNODE, NodeNrD2);
        buildIn_D1_Part(S1, posOR_D1, SLPUSE, SLNODE, NodeNrD2);
        buildIn_D2_Part(S1, posOR_D2, SLPUSE, SLNODE, NodeNrD2);
}


void uwggraphC::buildIn_D1_Part(SliceC & S1, int posOR_D1, SliceC::iterator SLPUSE,
SliceC::iterator SLNODE, int D2_RefNr) {
        int pos1 = 0;
        int ret = 0;
        int KnotenNr1 = 0;
        int LineNr1 = 0;
        SliceC::iterator ITER1 = find_D1_Node(S1, SLPUSE, SLNODE);
        SliceC::iterator HELP = ITER1;
        switch(ITER1->first.getKFGNodeType()) {
        case IFC:
                HELP = find_D1_Node(S1, ITER1, SLNODE);
                ret = buildInIfTree(S1,ITER1,HELP,posOR_D1);
                break;
        case LC:
                HELP = find_D1_Node(S1, ITER1, SLNODE);
                buildInLoopTree(S1,posOR_D1,ITER1,SLNODE);
                break;
        default:
                addAllD1Nodes(S1,ITER1,posOR_D1,D2_RefNr);
        }
}
```

```
SliceC::iterator uwggraphC::find_D1_Node(SliceC &  S1, SliceC::iterator SLUSE,
SliceC::iterator SLNODE) {
        int helpNr1 = 0;
        int helpNr2 = 0;
        int dummy1 = 0;
        int dummy2 = 0;
        int KnotenNrRet = 0;
        int KnotenNrRet1 = 0;
        int PUseNr = SLPUSE->first.getKnotenNummer();
        int PUseLevel = SLPUSE->first.getLevel();
        SliceC::iterator RETURNITER = SLNODE;
        KnotenNrRet1 = SLNODE->first.getKnotenNummer();
        if (KnotenNrRet1 == (PUseNr + 1)) {
                RETURNITER = defineIterator(S1,KnotenNrRet1);
                dummy1 = 1;
                dummy2 = 1;
        }
        SliceC::Nachfolger::iterator BEGIN1 = SLNODE->second.begin();
        SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
        while ((BEGIN1 != END1) && (dummy1 != 1)) {
                helpNr1 = BEGIN1->first;
                if ((S1[helpNr1].first.getKnotenNummer() > PUseNr) &&
(S1[helpNr1].first.getLevel() >= PUseLevel)) {
                        if (S1[helpNr1].first.getLevel() > PUseLevel) {
                                KnotenNrRet1 = S1[helpNr1].first.getKnotenNummer();
                                if (KnotenNrRet1 == (PUseNr + 1)) {
                                        RETURNITER = defineIterator(S1,KnotenNrRet1);
                                        dummy1 = 1;
                                        dummy2 = 1;
                                }
                                SliceC::Nachfolger::iterator BEGIN2 =
S1[helpNr1].second.begin();
                                SliceC::Nachfolger::iterator END2 =
S1[helpNr1].second.end();
                                while ((BEGIN2 != END2) && (dummy2 != 1)) {
                                        if (BEGIN2->second == KFK) {
                                                helpNr2 = BEGIN2->first;
                                                KnotenNrRet =
S1[helpNr2].first.getKnotenNummer();
                                                if (KnotenNrRet != PUseNr) {
                                                        RETURNITER =
defineIterator(S1,KnotenNrRet);
                                                        dummy1 = 1;
                                                        dummy2 = 1;
                                                }
                                        }
                                        BEGIN2++;
                                }
                        }
                        else {
                                KnotenNrRet = S1[helpNr1].first.getKnotenNummer();
                                RETURNITER = defineIterator(S1,KnotenNrRet);
                                dummy1 = 1;
                        }
                }
                BEGIN1++;
        }
        if ((RETURNITER == SLNODE) && (dummy1 == 0)) {
                BEGIN1 = SLNODE->second.begin();
                while (BEGIN1 !=  END1) {
                        helpNr1 = BEGIN1->first;
                        KnotenNrRet = S1[helpNr1].first.getKnotenNummer();
                        SliceC::iterator SLNODE = defineIterator(S1,KnotenNrRet);
                        RETURNITER = find_D1_Node(S1, SLPUSE, SLNODE);
                        BEGIN1++;
                }
        }
        return RETURNITER;
}

void uwggraphC::buildIn_D2_Part(SliceC & S1, int posOR_D2, SliceC::iterator SLPUSE,
SliceC::iterator SLNODE, int KnotenNrD2) {
        int pos1 = 0;
        int pos2 = 0;
        int ret = 0;
```

```
int helpNr1 = 0;
int helpNr2 = 0;
```

```
        int KnotenNr1 = 0;
        int LineNr1 = 0;
        int KnotenNrD2Ret = 0;
        int USEinD2KnotenNr = 0;
        char Bemerkung1[1000];
        char NodeText1[128];
        SliceC::iterator D2_NODE = defineIterator(S1, KnotenNrD2);
        KnotenNr1 = D2_NODE->first.getKnotenNummer();
        LineNr1 = D2_NODE->first.getLineNr();
        sprintf(NodeText1,"Op%d",KnotenNr1);
        //sprintf(Bemerkung1,"Line %d",LineNr1);
        strcpy(Bemerkung1,D2_NODE->first.getCodeLine());
        uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText1,Bemerkung1);
        pos1 = insert(hknoten1);
        verbindeEcken(posOR_D2,pos1,0);
        SliceC::Nachfolger::iterator BEGIN1 = D2_NODE->second.begin();
        SliceC::Nachfolger::iterator END1 = D2_NODE->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == DFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        LineNr1 = S1[helpNr1].first.getLineNr();
                        sprintf(NodeText1,"Op%d",KnotenNr1);
                        //sprintf(Bemerkung1,"Line %d",LineNr1);
                        strcpy(Bemerkung1,S1[helpNr1].first.getCodeLine());
                        uwgknotenC hknoten2(CAUSE,KnotenNr1,NodeText1,Bemerkung1);
                        pos2 = insert(hknoten2);
                        verbindeEcken(posOR_D2,pos2,0);
                        SliceC::Nachfolger::iterator BEGIN2 = S1[helpNr1].second.begin();
                        SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                        while (BEGIN2 != END2) {
                                helpNr2 = BEGIN2->first;
                                if ((BEGIN2->second == DFK) &&
(S1[helpNr2].first.getKnotenNummer() != KnotenNrD2)) {
                                        //SliceC::iterator SLNODE1 =
defineIterator(S1,KnotenNr1);
                        //buildIn_D2_Part(S1,posOR,SLPUSE,SLNODE1,KnotenNrD2);
                                }
                                BEGIN2++;
                        }
                }
                BEGIN1++;
        }
        SliceC::iterator USEinD2 = find_D1_Node(S1, SLPUSE, SLNODE);
        SliceC::iterator HELP = USEinD2;
        if (USEinD2->first.getLevel() > SLPUSE->first.getLevel()) {
                USEinD2KnotenNr = USEinD2->first.getKnotenNummer();
                KnotenNrD2Ret = findD2Node(S1, USEinD2);
                if (KnotenNrD2Ret == USEinD2KnotenNr) {
                        switch(USEinD2->first.getKFGNodeType()) {
                        case IFC:
                                HELP = find_D1_Node(S1,USEinD2,SLNODE);
                                //ret = buildInIfTree(S1,USEinD2,HELP,posOR);
                                ret = buildD2InIfTree(S1,USEinD2,HELP,posOR_D2);
                                break;
                        case LC:
                                HELP = find_D1_Node(S1,USEinD2,SLNODE);
                                buildInLoopTree(S1,posOR_D2,USEinD2,SLNODE);
                                break;
                        default:
                                cout << "Oups!!!" << endl;
                        }
                }
        }
}

int uwggraphC::findD2Node(SliceC & S1, SliceC::iterator PUSE) {
        int dummy = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int KnotenNrReturn = PUSE->first.getKnotenNummer();
        int PUSEKnotenNr = PUSE->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN1 = PUSE->second.begin();
        SliceC::Nachfolger::iterator END1 = PUSE->second.end();
        while ((BEGIN1 != END1) && (dummy != 1)) {
```

```
if (BEGIN1->second == DFK) {
        helpNr1 = BEGIN1->first;
```

```
                        SliceC::Nachfolger::iterator BEGIN2 =
S1[helpNr1].second.begin();
                        SliceC::Nachfolger::iterator END2 =
S1[helpNr1].second.end();
                        while ((BEGIN2 != END2) && (dummy != 1)){
                                if (BEGIN2->second == KFK) {
                                        helpNr2 = BEGIN2->first;
                                        if (S1[helpNr2].first.getKnotenNummer() ==
PUSEKnotenNr) {
                                                KnotenNrReturn =
S1[helpNr2].first.getKnotenNummer();
                                                dummy = 1;
                                        }
                                }
                                BEGIN2++;
                        }
                }
                BEGIN1++;
        }
        return KnotenNrReturn;
}

int uwggraphC::buildIn_LoopKF_ANDPart(SliceC & S1, int posAND_KF, SliceC::iterator SLPUSE) {
        int pos = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int RefPUse = 0;
        int KnotenNr = 0;
        int LineNr = 0;
        int dummy = 0;
        int dummy1 = 0;
        char Bemerkung[1000];
        char NodeText[128];
        RefPUse = SLPUSE->first.getKnotenNummer();
        SliceC::Nachfolger::iterator BEGIN = SLPUSE->second.begin();
        SliceC::Nachfolger::iterator END = SLPUSE->second.end();
        while ((BEGIN != END) && (dummy1 != 1)) {
                if (BEGIN->second == DFK) {
                        helpNr1 = BEGIN->first;
                        SliceC::Nachfolger::iterator BEGIN2 = S1[helpNr1].second.begin();
                        SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                        while ((BEGIN2 != END2) && (dummy != 1)) {
                                if (BEGIN2->second == KFK) {
                                        helpNr2 = BEGIN2->first;
                                        if (S1[helpNr2].first.getKnotenNummer() == RefPUse) {
                                                KnotenNr = S1[helpNr1].first.getKnotenNummer();
                                                LineNr = S1[helpNr1].first.getLineNr();
                                                sprintf(NodeText,"Op%d",KnotenNr);
                                                //sprintf(Bemerkung,"Line %d",LineNr);

        strcpy(Bemerkung,S1[helpNr1].first.getCodeLine());
                                                uwgknotenC
hknoten(CAUSE,KnotenNr,NodeText,Bemerkung);
                                                pos = insert(hknoten);
                                                verbindeEcken(posAND_KF,pos,0);
                                                dummy = 1;
                                                dummy1 = 1;
                                        }
                                }
                                BEGIN2++;
                        }
                }
                BEGIN++;
        }
        return KnotenNr;
}

void uwggraphC::buildIn_LoopKF_ORPart(SliceC & S1, int posLOOP_KF, SliceC::iterator
SL1, int KnotenNrD2) {
        int pos1 = 0;
        int pos2 = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int KnotenNr = 0;
        int KnotenNr1 = 0;
        int LineNr = 0;
        int LineNr1 = 0;
```

```
        char NodeText[128];
        char Bemerkung[1000];
        KnotenNr = SL1->first.getKnotenNummer();
        LineNr = SL1->first.getLineNr();
        sprintf(NodeText,"Op%d",KnotenNr);
        //sprintf(Bemerkung,"Line %d",LineNr);
        strcpy(Bemerkung,SL1->first.getCodeLine());
        uwgknotenC hknoten1(CAUSE, KnotenNr, NodeText,Bemerkung);
        pos1 = insert(hknoten1);
        verbindeEcken(posLOOP_KF,pos1,0);
        SliceC::Nachfolger::iterator BEGIN = SL1->second.begin();
        SliceC::Nachfolger::iterator END = SL1->second.end();
        while (BEGIN != END) {
                if (BEGIN->second == DFK) {
                        helpNr1 = BEGIN->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        if (KnotenNr1 != KnotenNrD2) {
                                LineNr1 = S1[helpNr1].first.getLineNr();
                                sprintf(NodeText,"Op%d",KnotenNr1);
                                //sprintf(Bemerkung,"Line %d",LineNr1);
                                strcpy(Bemerkung,SL1->first.getCodeLine());
                                uwgknotenC hknoten1(CAUSE, KnotenNr1, NodeText,Bemerkung);
                                pos1 = insert(hknoten1);
                                verbindeEcken(posLOOP_KF,pos1,0);
                        }
                        SliceC::iterator NEXTNODE = defineIterator(S1,KnotenNr1);
                        SliceC::Nachfolger::iterator BEGIN2 = NEXTNODE->second.begin();
                        SliceC::Nachfolger::iterator END2 = NEXTNODE->second.end();
                        while (BEGIN2 != END2) {
                                if (BEGIN2->second == DFK) {
                                        helpNr2 = BEGIN2->first;
                                        if (NEXTNODE->first.getKnotenNummer() != KnotenNrD2) {
                                                buildIn_LoopKF_ORPart(S1, posLOOP_KF,
NEXTNODE, KnotenNrD2);
                                        }
                                }
                                BEGIN2++;
                        }
                }
                BEGIN++;
        }
}

void uwggraphC::addAllD1Nodes(SliceC & S1, SliceC::iterator SLNODE, int posGatter,
int D2_RefNr) {
        int pos1 = 0;
        int ret = 0;
        int checkNr = 1;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int SliceNr2 = 0;
        int KnotenNr0 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int LineNr1 = 0;
        int LineNr0 = 0;
        char Bemerkung1[1000];
        char NodeText1[128];
        KnotenNr0 = SLNODE->first.getKnotenNummer();
        LineNr0 = SLNODE->first.getLineNr();
        sprintf(NodeText1,"Op%d",KnotenNr0);
        //sprintf(Bemerkung1,"Line %d",LineNr0);
        strcpy(Bemerkung1,SLNODE->first.getCodeLine());
        uwgknotenC hknoten1(CAUSE,KnotenNr0,NodeText1,Bemerkung1);
        pos1 = insert(hknoten1);
        verbindeEcken(posGatter,pos1,0);
        SliceC::Nachfolger::iterator BEGIN1 = SLNODE->second.begin();
        SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
        while (BEGIN1 != END1) {
                if (BEGIN1->second == DFK) {
                        helpNr1 = BEGIN1->first;
                        KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
                        SLNODE = defineIterator(S1,KnotenNr1);
                        if ((KnotenNr1 != D2_RefNr) && (KnotenNr1 != KnotenNr0)) {
                                SliceC::Nachfolger::iterator BEGIN2 = S1[helpNr1].second.begin();
```

```
                                SliceC::Nachfolger::iterator END2 = S1[helpNr1].second.end();
                                while (BEGIN2 != END2) {
                                        if (BEGIN2->second == KFK) {
                                                helpNr2 = BEGIN2->first;
                                                SliceNr2 =
S1[helpNr2].first.getKnotenNummer();
                                                SliceC::iterator HELPPUSE =
defineIterator(S1, SliceNr2);
                                                checkNr = checkUWGNode(SliceNr2);
                                                if (checkNr == 0) {
                                                        SliceC::iterator HELP1 =
findOutmostPUse(S1,HELPPUSE);
                                                        switch (HELP1-
>first.getKFGNodeType()) {
                                                        case LC:
                                                                buildInLoopTree(S1,
posGatter, HELPPUSE, SLNODE);
                                                                break;
                                                        case IFC:
                                                                ret =
buildInIfTree(S1,HELPPUSE,SLNODE,posGatter);
                                                                break;
                                                        default:
                                                                cout << "Incorrect
path! " << endl;
                                                        }
                                                }
                                        }
                                        BEGIN2++;
                                }
                                if (checkNr == 1) {
                                        LineNr1 = S1[helpNr1].first.getLineNr();
                                        sprintf(NodeText1,"Op%d",KnotenNr1);
                                        //sprintf(Bemerkung1,"Line %d",LineNr1);
                                        strcpy(Bemerkung1,S1[helpNr1].first.getCodeLine());
                                        uwgknotenC
hknoten2(CAUSE,KnotenNr1,NodeText1,Bemerkung1);
                                        pos1 = insert(hknoten2);
                                        verbindeEcken(posGatter,pos1,0);
                                        addAllD1Nodes(S1,SLNODE,posGatter,D2_RefNr);
                                }
                        }
                }
                BEGIN1++;
        }
}

int uwggraphC::buildD2InIfTree(SliceC & S1, SliceC::iterator SLIF, SliceC::iterator
SLORIG, int pos1) {
        int pos2 = 0;
        int posIFOR = 0;
        int posIFDFOR = 0;
        int posIFKFOR = 0;
        int posIFDFORA2 = 0;
        char NodeText[128];
        char Bemerkung[1000];
        int KnotenNr = SLIF->first.getKnotenNummer();
        int LineNr = SLIF->first.getLineNr();
        sprintf(NodeText,"Verzweigung_D+(%d)",KnotenNr);
        strcpy(Bemerkung,SLIF->first.getCodeLine());
        uwgknotenC hknoten1(OR,NodeText,Bemerkung);
        pos2 = insert(hknoten1);
        posIFOR = size() - 1;
        verbindeEcken(pos1, pos2,0);
        sprintf(NodeText,"Verzweigung_KF_D+(%d)",LineNr);
        uwgknotenC hknoten5(OR,NodeText);
        posIFKFOR = insert(hknoten5);
        verbindeEcken(pos2,posIFKFOR,0);
        buildD2_IFKF_Part(S1, posIFKFOR, SLIF);
        switch (SLORIG->first.getKnotenIdent()) {
        case THEN:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.1_D+(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.1_D+(%d)",LineNr);
```

```
sprintf(NodeText,"Alt.1_D+(%d)",LineNr);
uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
```

```
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.1_D+(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFOR = size() - 1;
                buildD2_A1_Part(S1, posIFDFOR, SLIF, SLORIG);
                }
                break;
        case ELSE:
                {
                sprintf(NodeText,"Verzweigung_DF_Alt.2_D+(%d)",LineNr);
                uwgknotenC hknoten2(AND,NodeText);
                insert(hknoten1,hknoten2,0);
                sprintf(Bemerkung,"Alt.2_D+(%d)",LineNr);
                sprintf(NodeText,"Alt.2_D+(%d)",LineNr);
                uwgknotenC hknoten3(CAUSE,NodeText,Bemerkung);
                insert(hknoten2,hknoten3,0);
                sprintf(NodeText,"Alt.2_D+(%d)",LineNr);
                uwgknotenC hknoten4(OR,NodeText);
                insert(hknoten2,hknoten4,0);
                posIFDFORA2 = size() - 1;
                buildD2_A1_Part(S1, posIFDFORA2, SLIF, SLORIG);
                }
                break;
        default:
                cout << "Ouch!!!" << endl;
        }
        return posIFDFOR;
}

void uwggraphC::buildD2_A1_Part(SliceC & S1, int posIFDFOR_D2, SliceC::iterator
SLIF, SliceC::iterator SLNODE) {
        int pos0_A1 = 0;
        int pos1_A1 = 0;
        int helpNr0 = 0;
        int helpNr1 = 0;
        int KnotenNr0 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int LineNr0 = 0;
        int LineNr1 = 0;
        char NodeText0[128];
        char Bemerkung0[1000];
        KnotenNr0 = SLNODE->first.getKnotenNummer();
        LineNr0 = SLNODE->first.getLineNr();
        sprintf(NodeText0,"Op%d",KnotenNr0);
        //sprintf(Bemerkung0,"Line %d",LineNr0);
        strcpy(Bemerkung0,SLNODE->first.getCodeLine());
        uwgknotenC hknoten0(CAUSE,KnotenNr0,NodeText0,Bemerkung0);
        pos0_A1 = insert(hknoten0);
        verbindeEcken(posIFDFOR_D2,pos0_A1,0);
        SliceC::Nachfolger::iterator BEGIN0 = SLNODE->second.begin();
        SliceC::Nachfolger::iterator END0 = SLNODE->second.end();
        while (BEGIN0 != END0) {
                helpNr0 = BEGIN0->first;
                KnotenNr1 = S1[helpNr0].first.getKnotenNummer();
                if (KnotenNr1 > KnotenNr0) {
                        LineNr1 = S1[helpNr0].first.getLineNr();
                        sprintf(NodeText0,"Op%d",KnotenNr1);
                        //sprintf(Bemerkung0,"Line %d",LineNr1);
                        strcpy(Bemerkung0,S1[helpNr0].first.getCodeLine());
                        uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText0,Bemerkung0);
                        pos1_A1 = insert(hknoten1);
                        verbindeEcken(posIFDFOR_D2,pos1_A1,0);
                        SliceC::Nachfolger::iterator BEGIN1 =
S1[helpNr0].second.begin();
                        SliceC::Nachfolger::iterator END1 =
S1[helpNr0].second.end();
                        while (BEGIN1 != END1) {
                                if (BEGIN1->second == DFK) {
                                        helpNr1 = BEGIN1->first;
                                        KnotenNr2 =
S1[helpNr1].first.getKnotenNummer();
                                        if (KnotenNr1 != KnotenNr2) {
```

```
                                        SliceC::iterator SLNODE =
    defineIterator(S1,KnotenNr2);
                                        buildD2_A1_Part(S1, posIFDFOR_D2,
5   SLIF, SLNODE);
                                    }
                                }
```

```
                                BEGIN1++;
                        }
                }
                BEGIN0++;
        }
}

void uwggraphC::buildD2_IFKF_Part(SliceC & S1, int posKFOR_D2, SliceC::iterator
SLPUSE) {
        int pos0_D2 = 0;
        int pos1_D2 = 0;
        int ret = 0;
        int dummy = 0;
        int helpNr0 = 0;
        int helpNr1 = 0;
        int helpNr2 = 0;
        int checkPUse = 0;
        int checkNode = 0;
        int KnotenNr0 = 0;
        int KnotenNr1 = 0;
        int KnotenNr2 = 0;
        int KnotenNr3 = 0;
        int LineNr0 = 0;
        int LineNr1 = 0;
        char NodeText0[128];
        char Bemerkung0[1000];
        KnotenNr0 = SLPUSE->first.getKnotenNummer();
        LineNr0 = SLPUSE->first.getLineNr();
        sprintf(NodeText0,"Op%d",KnotenNr0);
        //sprintf(Bemerkung0,"Line %d",LineNr0);
        strcpy(Bemerkung0,SLPUSE->first.getCodeLine());
        uwgknotenC hknoten0(CAUSE,KnotenNr0,NodeText0,Bemerkung0);
        pos0_D2 = insert(hknoten0);
        verbindeEcken(posKFOR_D2,pos0_D2,0);
        SliceC::Nachfolger::iterator BEGIN0 = SLPUSE->second.begin();
        SliceC::Nachfolger::iterator END0 = SLPUSE->second.end();
        while (BEGIN0 != END0) {
                helpNr0 = BEGIN0->first;
                KnotenNr1 = S1[helpNr0].first.getKnotenNummer();
                SliceC::iterator SLNODE = defineIterator(S1,KnotenNr1);
                if (KnotenNr1 > KnotenNr0) {
                        LineNr1 = S1[helpNr0].first.getLineNr();
                        sprintf(NodeText0,"Op%d",KnotenNr1);
                        //sprintf(Bemerkung0,"Line %d",LineNr1);
                        strcpy(Bemerkung0,S1[helpNr0].first.getCodeLine());
                        uwgknotenC hknoten1(CAUSE,KnotenNr1,NodeText0,Bemerkung0);
                        pos1_D2 = insert(hknoten1);
                        verbindeEcken(posKFOR_D2,pos1_D2,0);
                        SliceC::Nachfolger::iterator BEGIN1 = SLNODE-
>second.begin();
                        SliceC::Nachfolger::iterator END1 = SLNODE->second.end();
                        while (BEGIN1 != END1) {
                                if (BEGIN1->second == DFK) {
                                        helpNr1 = BEGIN1->first;
                                        KnotenNr2 =
S1[helpNr1].first.getKnotenNummer();
                                        if (KnotenNr2 != KnotenNr1) {
                                                SliceC::iterator SLNODE =
defineIterator(S1,KnotenNr2);
                                                SliceC::Nachfolger::iterator BEGIN2 =
S1[helpNr1].second.begin();
                                                SliceC::Nachfolger::iterator END2 =
S1[helpNr1].second.end();
                                                while (BEGIN2 != END2) {
                                                        if (BEGIN2->second == KFK) {
                                                                helpNr2 = BEGIN2-
>first;
                                                                KnotenNr3 =
S1[helpNr2].first.getKnotenNummer();
                                                                SliceC::iterator
HELPPUSE = defineIterator(S1,KnotenNr3);
                                                                if (SLPUSE-
>first.getKnotenNummer() > KnotenNr3) {
                                                                        checkPUse =
checkPUses1(S1,SLPUSE,HELPPUSE);
```

```
                                                    }
                                                    else {
                                                           checkPUse =
      checkPUses2(S1,SLPUSE,HELPPUSE);
5                                                   }
```

```
                                                        checkNode =
checkUWGNode(KnotenNr2);
                                                        if (checkNode == 0) {
                                                                if (checkPUse
== 0) {
        //HELPPUSE = findOutmostPUse(S1,HELPPUSE);
                                                                        switch
(HELPPUSE->first.getKFGNodeType()) {
                                                                        case LC:
        buildInLoopTree(S1, posKFOR_D2, HELPPUSE, SLNODE);
        dummy = 1;
        break;
                                                                        case
IFC:
        ret = buildInIfTree(S1, HELPPUSE, SLNODE, posKFOR_D2);
        dummy = 1;
        break;
                                                                        default:
        cout << "Incorrect path in A1! " << endl;
                                                                        }
                                                                }
                                                                else {
                                                                        LineNr1
= S1[helpNr2].first.getLineNr();
        sprintf(NodeText0,"Op%d",KnotenNr2);
        //sprintf(Bemerkung0,"Line %d",LineNr1);
        strcpy(Bemerkung0,S1[helpNr2].first.getCodeLine());
        uwgknotenC hknoten1(CAUSE,KnotenNr2,NodeText0,Bemerkung0);
                                                                        pos1_D2
= insert(hknoten1);
        verbindeEcken(posKFOR_D2,pos1_D2,0);
                                                                }
                                                        }
                                                }
                                        }
                                        BEGIN2++;
                                }
                        }
                }
                BEGIN1++;
        }
    }
    BEGIN0++;
  }
}

SliceC::iterator uwggraphC::findOutmostPUse(SliceC & S1, SliceC::iterator SLPUSE) {
    int dummy = 0;
    int helpNr1 = 0;
    int checkNr = 0;
    int KnotenNr1 = 0;
    SliceC::iterator RETURN = SLPUSE;
    SliceC::Nachfolger::iterator BEGIN1 = SLPUSE->second.begin();
    SliceC::Nachfolger::iterator END1 = SLPUSE->second.end();
    while ((BEGIN1 != END1) && (dummy != 1)){
        if (BEGIN1->second == KFK) {
            helpNr1 = BEGIN1->first;
            KnotenNr1 = S1[helpNr1].first.getKnotenNummer();
            checkNr = checkUWGNode(KnotenNr1);
            if (checkNr == 0) {
                SLPUSE = defineIterator(S1,KnotenNr1);
                RETURN = findOutmostPUse(S1, SLPUSE);
                dummy = 1;
```

```
                        }
                        else {
                                RETURN = SLPUSE;
                                dummy = 1;
                        }
                }
                BEGIN1++;
        }
```

```
        return RETURN;
}

int uwggraphC::checkUWGNode(int SliceNr) {
        int dummy = 0;
        uwggraphC::iterator ITER = begin();
        while (ITER != end()) {
                if ((*ITER).first.getCauseNr() == SliceNr) {
                        dummy = 1;
                }
                ITER++;
        }
        if (dummy == 1)
                return 1;
        else
                return 0;
}

SliceC::iterator uwggraphC::defineIterator(SliceC & S1, int SliceNr) {
        int dummy = 0;
        SliceC::iterator ITER = S1.begin();
        SliceC::iterator HELP = S1.begin();
        while ((ITER != S1.end()) && (dummy != 1)) {
                if ((*ITER).first.getKnotenNummer() == SliceNr) {
                        HELP = ITER;
                        dummy = 1;
                }
                ITER++;
        }
        return HELP;
}

void uwggraphC::SliceAusgeben(SliceC & S1) {
        //int a;
        ofstream Ziel("Slice.slc");
        ostream_iterator<KFGListNodeC> POSIT(Ziel, " Knoten\n");
        ostream_iterator<KFGListNodeC> POSIT2(Ziel, "        Nachfolger: ");
        ostream_iterator<GraphKanteC> KANTEIT(Ziel, "\n");
        Ziel << "SLICE:" << endl << endl;
        for (int i = 0; i < S1.size(); ++i) {
                Ziel << S1[i].first << " <";
                SliceC::Nachfolger::iterator IT = S1[i].second.begin();
                SliceC::Nachfolger::iterator ITEND = S1[i].second.end();
                while (IT != ITEND) {
                        Ziel << S1[(*IT).first].first << ' '      // Ecke
                << endl << "Kante:" << (*IT).second << ' ';   // Kantenwert
                        ++IT;
                }
                Ziel << ">\n";
        }
}

#include "uwgkante.h"
#include <iostream>

        ostream& operator << ( ostream& os, const uwgkanteC& Node){
                os << Node.Number << endl ;
                return os;
        }
```

```
#include "uwgknoten.h"
#include <iostream>

ostream& operator << ( ostream& os, const uwgknotenC& Node){
        //os << Node.GatterIdent << "    " << Node.CauseNr << " " <<
Node.gattertext << endl ;
        //return os;
        if (Node.getNodeIdent() == EFFECT) {
                os << "%%EFFECT:\"" << Node.gatterbemerkung << "\":\"" <<
Node.gattertext << "\"," << Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }
        else if (Node.getNodeIdent() == OR) {
                os << "%%OR:1:\"" << Node.gattertext << "\":\"none\"," <<
Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }
        else if (Node.getNodeIdent() == AND) {
                os << "%%AND:\"" << Node.gattertext << "\":\"none\"," <<
Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }
        else if (Node.getNodeIdent() == NOT) {
                os << "%%NOT:\"" << Node.gattertext << "\":\"none\"," <<
Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }
        /*else if (Node.getNodeIdent() == CAUSE) {
                os << "%%CAUSE:\"" << Node.gattertext << "\":\"none\"," <<
Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }*/
        else if (Node.getNodeIdent() == CAUSE) {
                os << "%%CAUSE:\"" << Node.gatterbemerkung << "\":\"" <<
Node.gattertext << "\"," << Node.GatterIdent <<  ",,/0;" << endl;
                return os;
        }
        else {
                os << Node.GatterIdent << "      " << Node.CauseNr << " " <<
Node.gattertext << endl ;
                return os;
        }
}


int uwgknotenC::DummyNr = 1;
```

The following publications are cited in this document:

[1] N. Leveson, Safety Verification of ADA Programs using software fault trees, IEEE Software, pages 48 to 59, July 1991

[2] Weiser M., *Program Slicing*, in: IEEE Transaction on Software Engineering, Vol. 10, No. 4, July 1984, pp. 352-357

[3] Liggesmeyer P., Modultest und Modulverifikation [Module Test and Module Verification] - State of the Art, Mannheim, Vienna, Zurich: BI Wissenschaftsverlag 1990

[4] DIN 25424-1: Fehlerbaumanalysen; Methoden und Bildzeichen [Fault Tree Analyses; Methods and Graphic Symbols], September 1981

[5] DIN 25424-2: Fehlerbaumanalyse; Handrechenverfahren zur Auswertung eines Fehlerbaums [Fault Tree Analysis; Manual Computation Methods for Evaluating a Fault Tree], Berlin, Beuth Verlag GmbH, April 1990

Patent Claims

1. A method for ascertaining an overall fault description for at least one section of a computer program, using a computer,
   - in which at least the section of the computer program is stored,
   - in which a control flow description is ascertained for the section of the computer program, which control flow description describes a flow of control information in the section of the computer program, and a data flow description is ascertained for the section of the computer program, which data flow description describes a flow of data in the section of the computer program, and said descriptions are combined into a joint flow description for the section of the computer program,
   - in which program elements are selected from the section of the computer program,
   - in which, for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
   - in which a fault description for a reference element describes possible faults in the respective reference element, and
   - in which the element fault descriptions are used to ascertain the overall fault description, with a structure of the overall fault description taking into account a structure of the joint flow description.

2. The method as claimed in claim 1,
in which the control flow description is in the form of a control flow graph.

3. The method as claimed in claim 1 or 2,
in which the data flow description is in the form of a data

flow graph.

4. The method as claimed in one of the preceding claims,

- in which the fault description is in the form of a stored fault tree,
- in which the element fault description is ascertained as an element fault tree, and
- in which the overall fault description is ascertained as an overall fault tree.

5. The method as claimed in one of the preceding claims,
used for fault analysis in the section of the computer program.

6. The method as claimed in one of the preceding claims,

- in which the overall fault description is ascertained as an overall fault tree,
- in which the overall fault tree is altered in terms of prescribable boundary conditions.

7. The method as claimed in claim 6,
in which the alteration is made by adding a complementary fault tree.

8. An arrangement for ascertaining an overall fault description for at least one section of a computer program, using a computer,
having a processor which is set up such that the following steps can be carried out:

- at least the section of the computer program is stored,
- a control flow description is ascertained for the section of the computer program, which control flow description describes a flow of control information in the section of the computer program, and a data flow description is ascertained for the section of the computer program, which data flow description describes a flow of data in the section of the computer program, and said descriptions are combined into a joint flow description for the section of the computer program,

ART 34 AMDT

- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element, and
- the element fault descriptions are used to ascertain the overall fault description, with a structure of the overall fault description taking into account a structure of the joint flow description.

9. The arrangement as claimed in claim 8,
in which the processor is set up such that the control flow description is in the form of a control flow graph.

10. The arrangement as claimed in claim 8 or 9,
in which the processor is set up such that the data flow description is in the form of a data flow graph.

11. The arrangement as claimed in one of claims 8 to 10,
in which the processor is set up such that
- the fault description is in the form of a stored fault tree,
- the element fault description is ascertained as an element fault tree, and
- the overall fault description is ascertained as an overall fault tree.

12. The arrangement as claimed in one of claims 8 to 11,
used for fault analysis in the section of the computer program.

13. The arrangement as claimed in one of claims 8 to 12,

in which the processor is set up so that

- the overall fault description is ascertained as an overall fault tree,
- the overall fault tree is altered in terms of prescribable boundary conditions.

14. The arrangement as claimed in claim 13,
in which the processor is set up such that the alteration is made by adding a complementary fault tree.

15. A computer program product comprising a computer-readable storage medium on which a program is stored which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description is ascertained for the section of the computer program, which control flow description describes a flow of control information in the section of the computer program, and a data flow description is ascertained for the section of the computer program, which data flow description describes a flow of data in the section of the computer program, and said descriptions are combined into a joint flow description for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element, and

ART 34 AMDT

- the element fault descriptions are used to ascertain the overall fault description, with a structure of the overall fault description taking into account a structure of the joint flow description.

16. A computer-readable storage medium on which a program is stored which, when it has been loaded into a memory in a computer, allows the computer to carry out the following steps for ascertaining an overall fault description for at least one section of a computer program:

- at least the section of the computer program is stored,
- a control flow description is ascertained for the section of the computer program, which control flow description describes a flow of control information in the section of the computer program, and a data flow description is ascertained for the section of the computer program, which data flow description describes a flow of data in the section of the computer program, and said descriptions are combined into a joint flow description for the section of the computer program,
- program elements are selected from the section of the computer program,
- for each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element,
- a fault description for a reference element describes possible faults in the respective reference element, and
- the element fault descriptions are used to

ascertain the overall fault description, with a structure of the overall fault description taking into account a structure of the joint flow description.

Abstract

Method and arrangement for ascertaining an overall fault description for at least one section of a computer program, and computer program product and computer-readable storage medium

The section of the computer program is used to ascertain a control flow description and a data flow description, and program elements are selected from the section of the computer program. For each selected program element, a stored fault description associated with a respective reference element is used to ascertain an element fault description which describes possible faults in the respective program element. The element fault descriptions are used to ascertain the overall fault description, taking into account the control flow description and the data flow description.

Figure 2

FIG 1

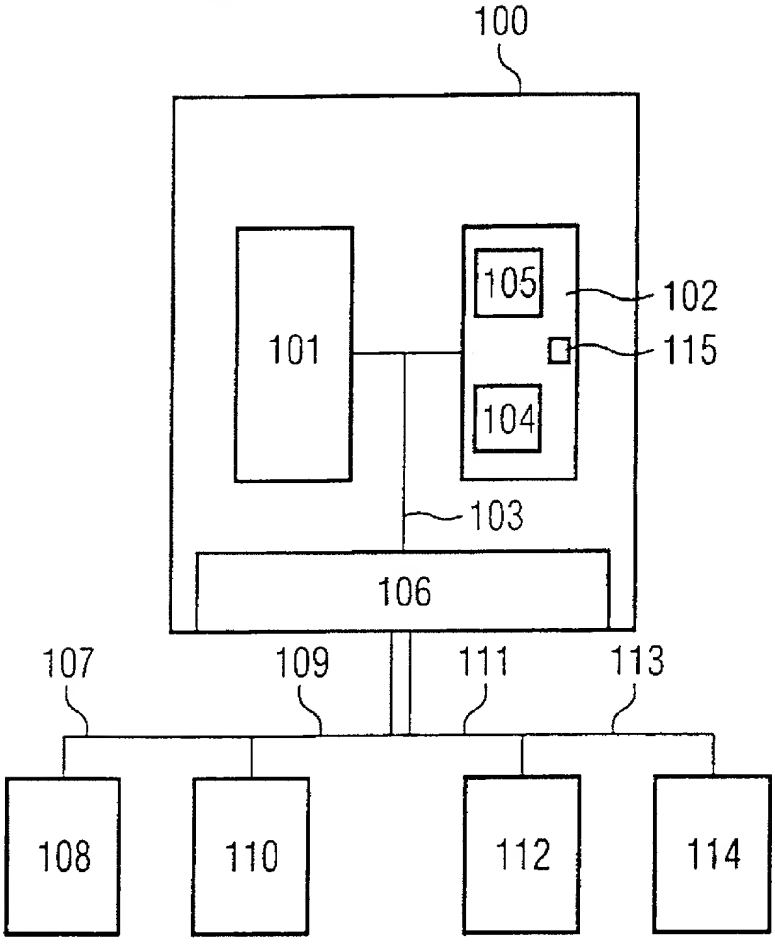


FIG 2

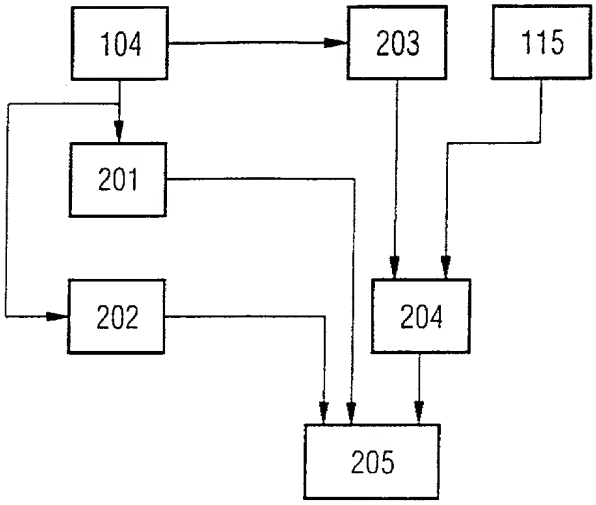


FIG 3

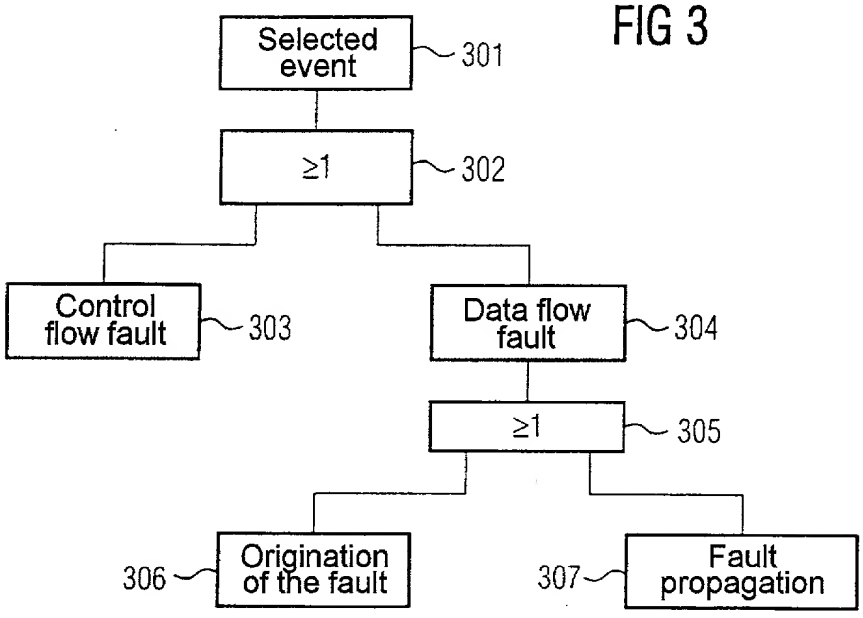


FIG 4A

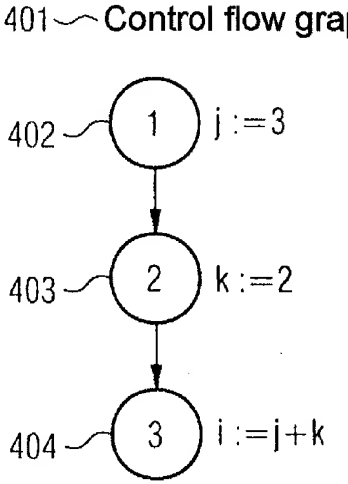


FIG 4B

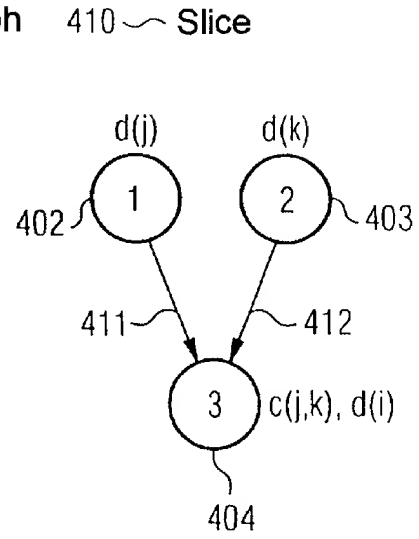


FIG 4C

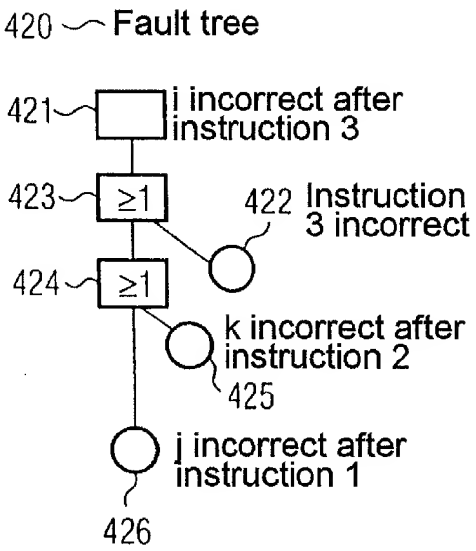


FIG 5A

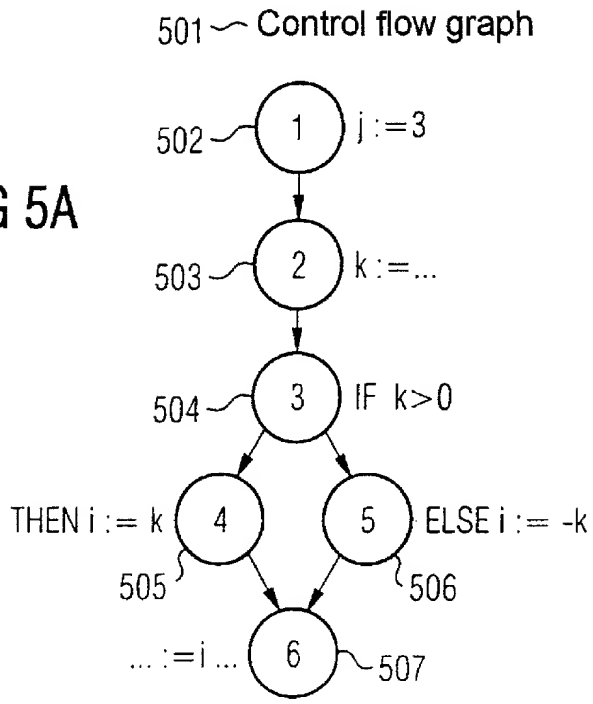

501 Control flow graph
502
1
j :=3
503
2
k :=...
504
3
IF k>0
THEN i := k
4
5
ELSE i := -k
505
506
... :=i ...
6
507

FIG 5B

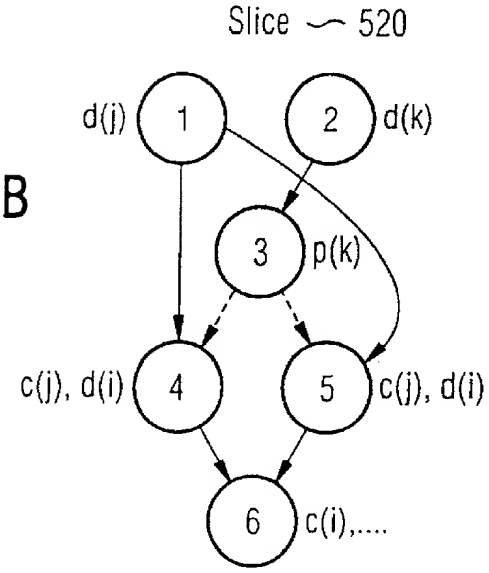

Slice 520
d(j)
1
2
d(k)
3
p(k)
c(j), d(i)
4
5
c(j), d(i)
6
c(i),....

FIG 6A

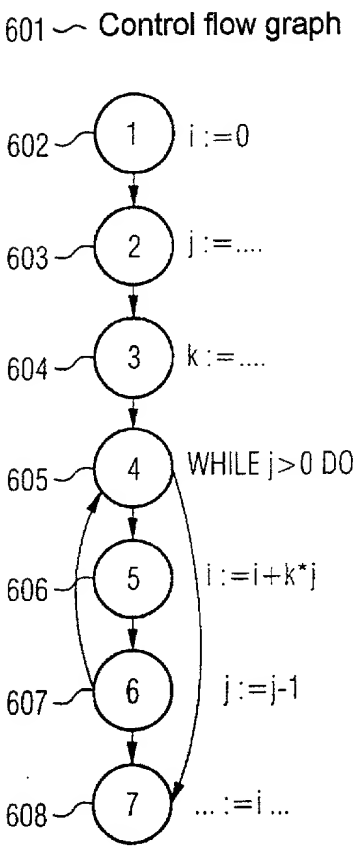


FIG 6B

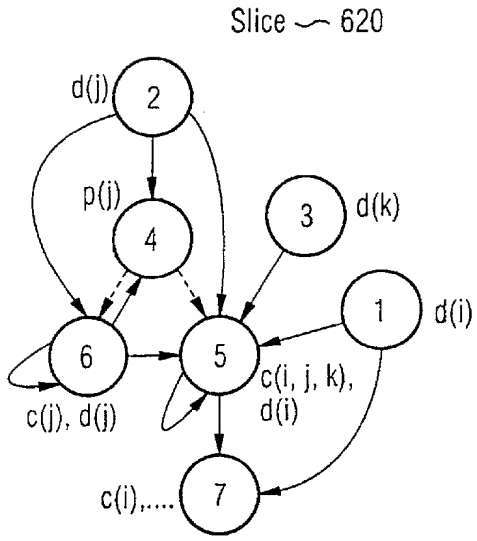


FIG 6C

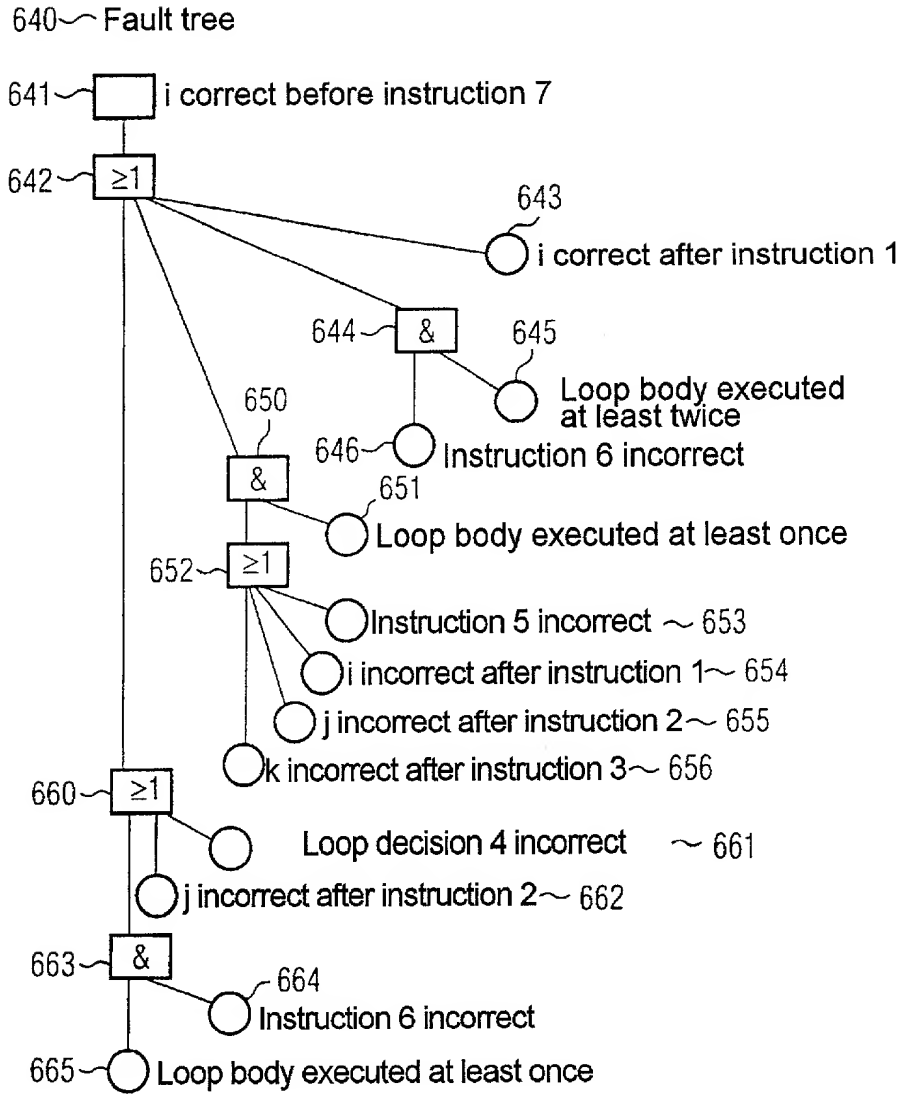

640 Fault tree
641
i correct before instruction 7
642
≥1
643
i correct after instruction 1
644
&
645
Loop body executed at least twice
650
646
Instruction 6 incorrect
&
651
Loop body executed at least once
652
≥1
Instruction 5 incorrect ~ 653
i incorrect after instruction 1~ 654
j incorrect after instruction 2~ 655
k incorrect after instruction 3~ 656
660
≥1
Loop decision 4 incorrect ~ 661
j incorrect after instruction 2~ 662
663
&
664
Instruction 6 incorrect
665
Loop body executed at least once

FIG 7

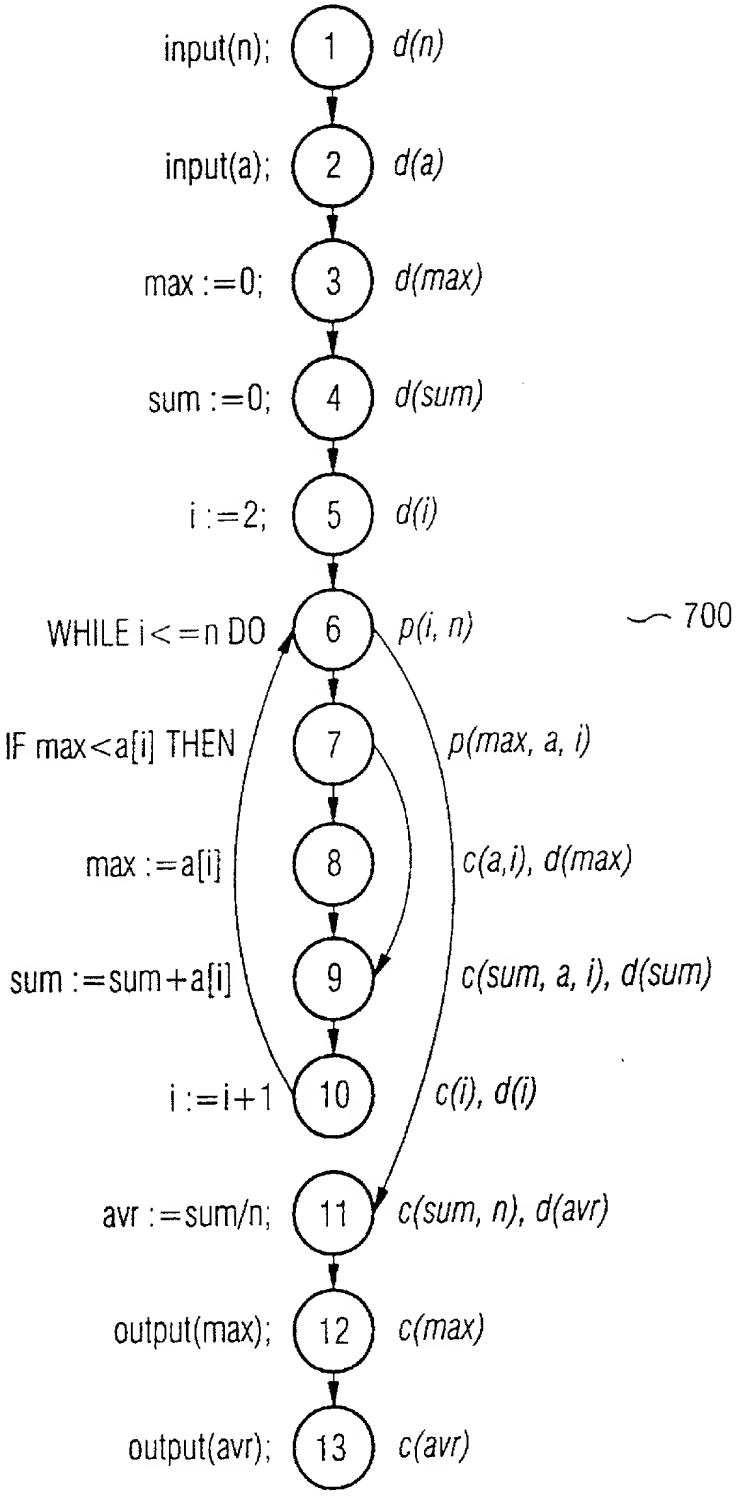

input(n);
1
d(n)
input(a);
2
d(a)
max :=0;
3
d(max)
sum :=0;
4
d(sum)
i :=2;
5
d(i)
WHILE i<=n DO
6
p(i, n)
700
IF max<a[i] THEN
7
p(max, a, i)
max :=a[i]
8
c(a,i), d(max)
sum :=sum+a[i]
9
c(sum, a, i), d(sum)
i :=i+1
10
c(i), d(i)
avr :=sum/n;
11
c(sum, n), d(avr)
output(max);
12
c(max)
output(avr);
13
c(avr)

FIG 8A

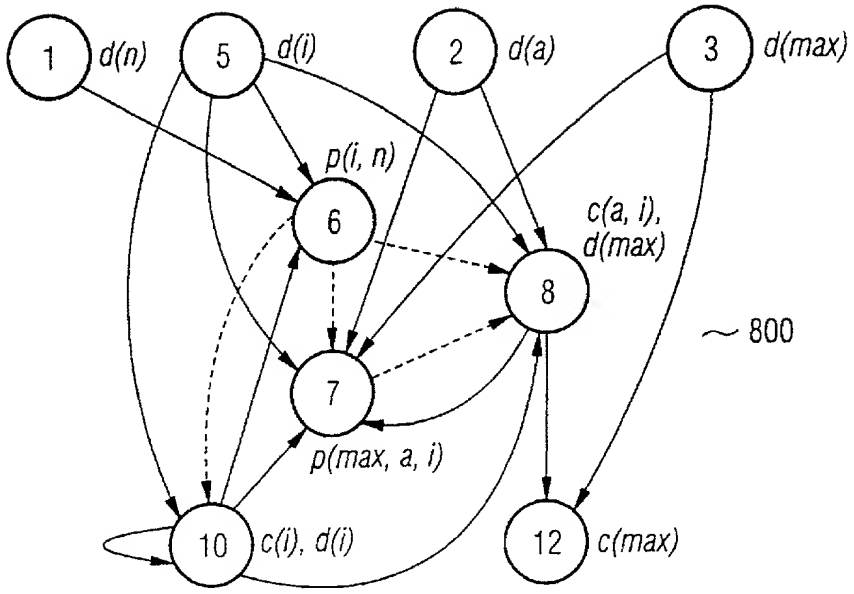

1
d(n)
5
d(i)
2
d(a)
3
d(max)
p(i, n)
6
c(a, i),
d(max)
8
~ 800
7
p(max, a, i)
10
c(i), d(i)
12
c(max)

FIG 8B

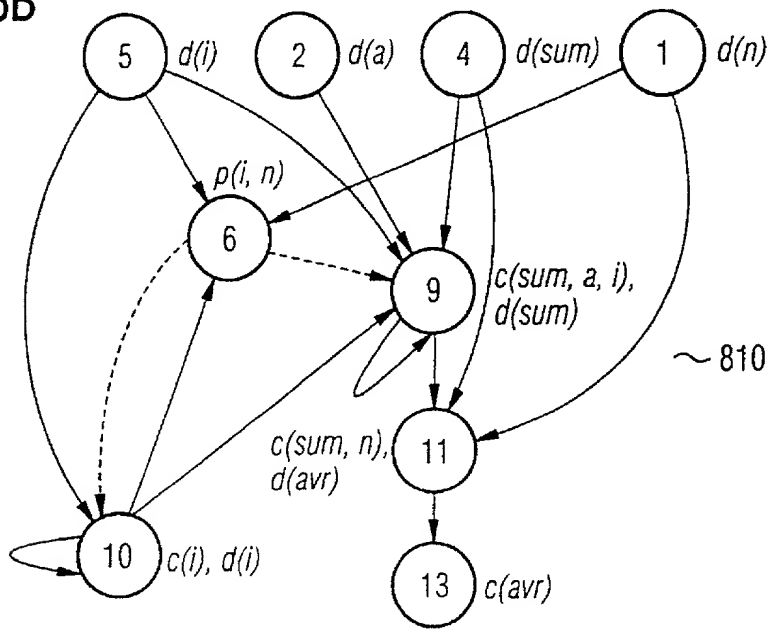

5
d(i)
2
d(a)
4
d(sum)
1
d(n)
p(i, n)
6
9
c(sum, a, i),
d(sum)
~ 810
c(sum, n),
d(avr)
11
10
c(i), d(i)
13
c(avr)

FIG 9

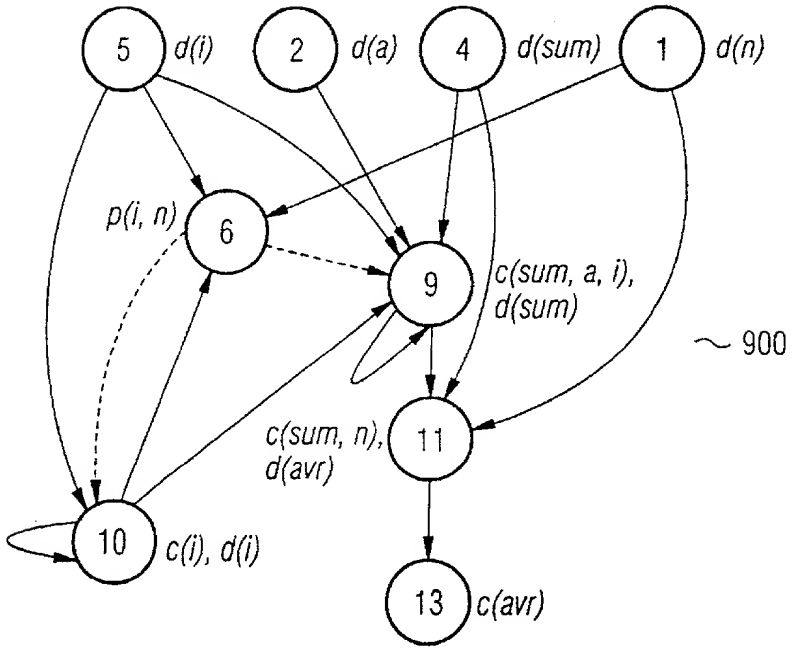

5
d(i)
2
d(a)
4
d(sum)
1
d(n)
p(i, n)
6
9
c(sum, a, i),
d(sum)
900
c(sum, n),
d(avr)
11
10
c(i), d(i)
13
c(avr)

FIG 10

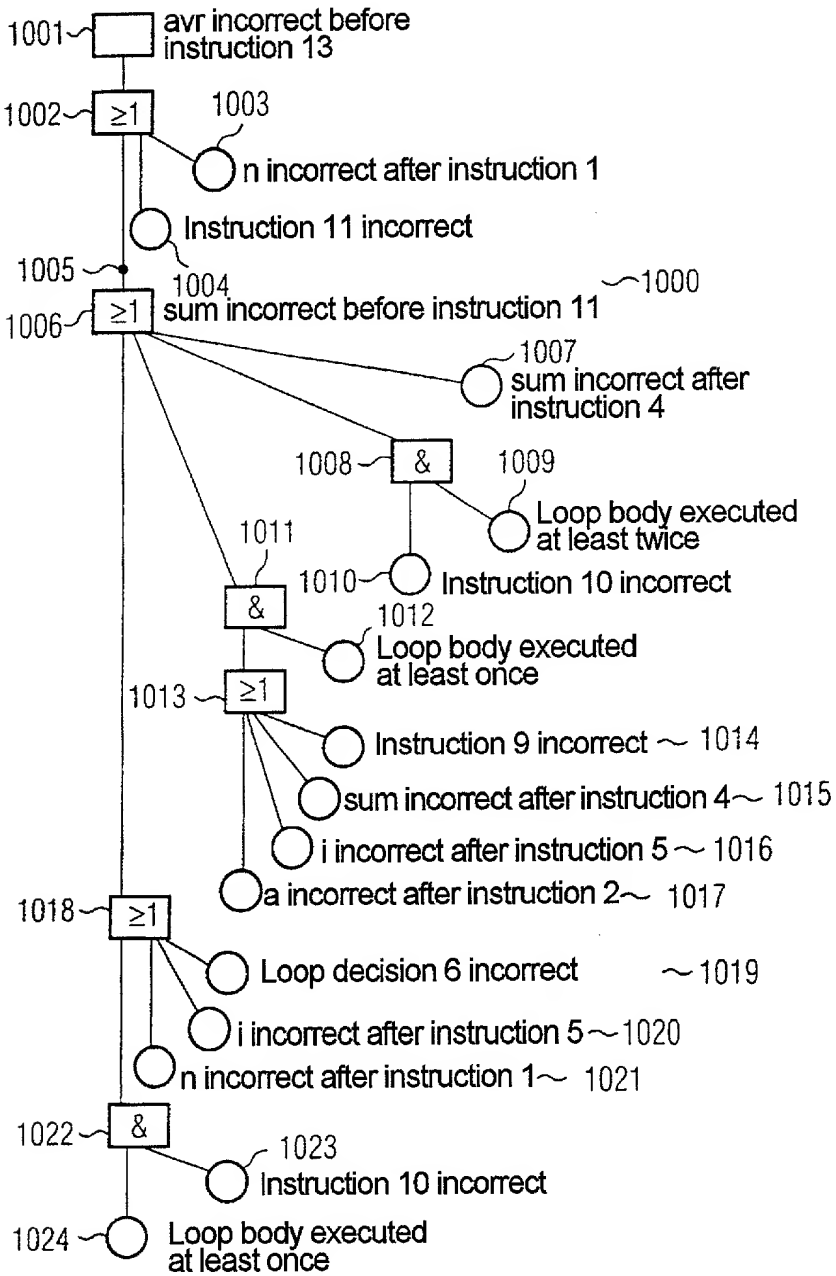
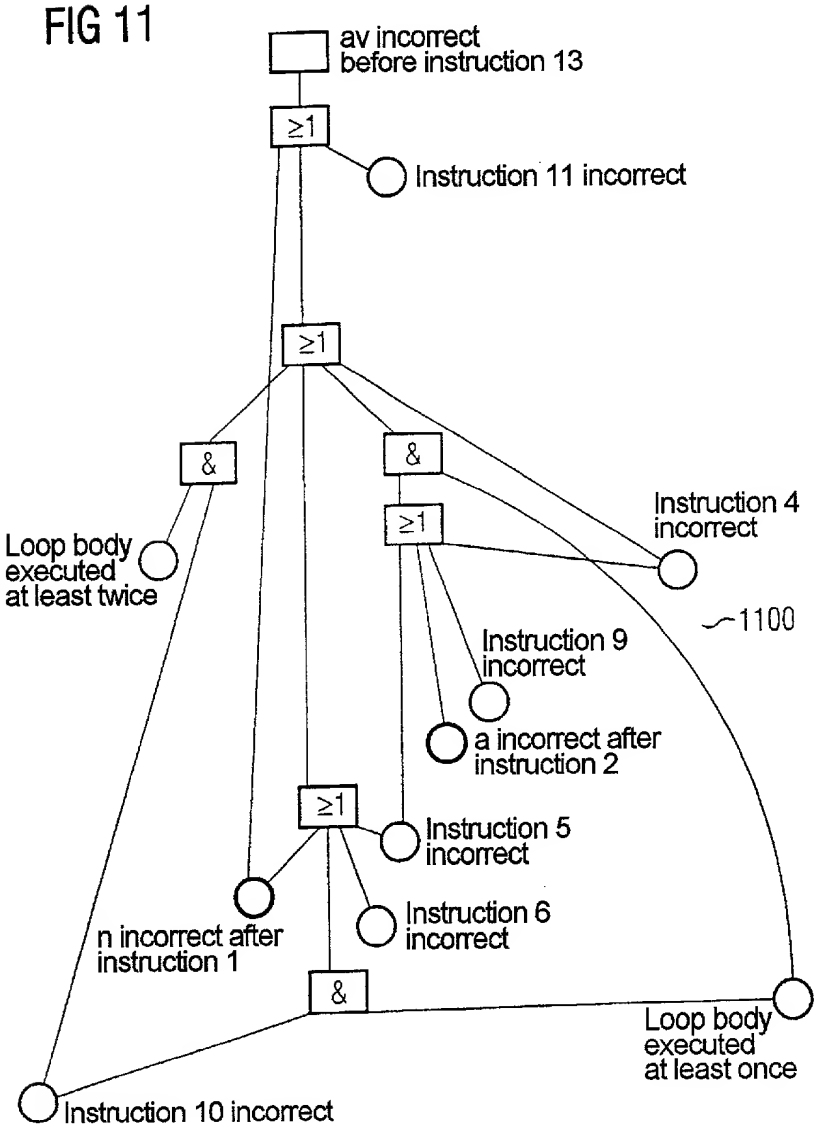

1001
avr incorrect before instruction 13
1002
≥1
1003
n incorrect after instruction 1
Instruction 11 incorrect
1004
1005
1006
≥1
sum incorrect before instruction 11
1000
1007
sum incorrect after instruction 4
1008
&
1009
Loop body executed at least twice
1011
1010
Instruction 10 incorrect
&
1012
Loop body executed at least once
1013
≥1
Instruction 9 incorrect 1014
sum incorrect after instruction 4 1015
i incorrect after instruction 5 1016
a incorrect after instruction 2 1017
1018
≥1
Loop decision 6 incorrect 1019
i incorrect after instruction 5 1020
n incorrect after instruction 1 1021
1022
&
1023
Instruction 10 incorrect
1024
Loop body executed at least once

FIG 11


av incorrect before instruction 13
≥1
Instruction 11 incorrect
≥1
&
&
≥1
Loop body executed at least twice
Instruction 4 incorrect
1100
Instruction 9 incorrect
a incorrect after instruction 2
≥1
Instruction 5 incorrect
n incorrect after instruction 1
Instruction 6 incorrect
&
Loop body executed at least once
Instruction 10 incorrect

Declaration and Power of Attorney For Patent Application
Erklärung Für Patentanmeldungen Mit Vollmacht
German Language Declaration

Als nachstehend benannter Erfinder erkläre ich hiermit an Eides Statt:

dass mein Wohnsitz, meine Postanschrift, und meine Staatsangehörigkeit den im Nachstehenden nach meinem Namen aufgeführten Angaben entsprechen,

dass ich, nach bestem Wissen der ursprüngliche, erste und alleinige Erfinder (falls nachstehend nur ein Name angegeben ist) oder ein ursprünglicher, erster und Miterfinder (falls nachstehend mehrere Namen aufgeführt sind) des Gegenstandes bin, für den dieser Antrag gestellt wird und für den ein Patent beantragt wird für die Erfindung mit dem Titel:

Verfahren und Anordnung zur Ermittlung einer Gesamtfehlerbeschreibung zumindest eines Teils eines Computerprogramms sowie Computerprogramm-Erzeugnis und computerlesbares Speichermedium

deren Beschreibung

(zutreffendes ankreuzen)
☐ hier beigefügt ist.
☒ am 03.04.2000 als
PCT internationale Anmeldung
PCT Anmeldungsnummer PCT/DE00/01001
eingereicht wurde und am _____
abgeändert wurde (falls tatsächlich abgeändert).

Ich bestätige hiermit, dass ich den Inhalt der obigen Patentanmeldung einschliesslich der Ansprüche durchgesehen und verstanden habe, die eventuell durch einen Zusatzantrag wie oben erwähnt abgeändert wurde.

Ich erkenne meine Pflicht zur Offenbarung irgendwelcher Informationen, die für die Prüfung der vorliegenden Anmeldung in Einklang mit Absatz 37, Bundesgesetzbuch, Paragraph 1.56(a) von Wichtigkeit sind, an.

Ich beanspruche hiermit ausländische Prioritätsvorteile gemäss Abschnitt 35 der Zivilprozessordnung der Vereinigten Staaten, Paragraph 119 aller unten angegebenen Auslandsanmeldungen für ein Patent oder eine Erfindersurkunde, und habe auch alle Auslandsanmeldungen für ein Patent oder eine Erfindersurkunde nachstehend gekennzeichnet, die ein Anmeldedatum haben, das vor dem Anmeldedatum der Anmeldung liegt, für die Priorität beansprucht wird.

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name,

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled

Method and arrangement for determining a total error description of at least one part of a computer programme and computer programme product and computer-readable storage medium

the specification of which

(check one)
☐ is attached hereto.
☒ was filed on 03.04.2000 as
PCT international application
PCT Application No. PCT/DE00/01001
and was amended on _____
(if applicable)

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, Code of Federal Regulations, §1.56(a).

I hereby claim foreign priority benefits under Title 35, United States Code, §119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

Form PTO-FB-240 (8-83)
Patent and Trademark Office-U.S. DEPARTMENT OF COMMERCE

German Language Declaration

Prior foreign appplications Priorität beansprucht			Priority Claimed	
19925239.4	DE	02.06.1999	☒	☐
(Number) (Nummer)	(Country) (Land)	(Day Month Year Filed) (Tag Monat Jahr eingereicht)	Yes Ja	No Nein
			☐	☐
(Number) (Nummer)	(Country) (Land)	(Day Month Year Filed) (Tag Monat Jahr eingereicht)	Yes Ja	No Nein
			☐	☐
(Number) (Nummer)	(Country) (Land)	(Day Month Year Filed) (Tag Monat Jahr eingereicht)	Yes Ja	No Nein

Ich beanspruche hiermit gemäss Absatz 35 der Zivilprozessordnung der Vereinigten Staaten, Paragraph 120, den Vorzug aller unten aufgeführten Anmeldungen und falls der Gegenstand aus jedem Anspruch dieser Anmeldung nicht in einer früheren amerikanischen Patentanmeldung laut dem ersten Paragraphen des Absatzes 35 der Zivilprozeßordnung der Vereinigten Staaten, Paragraph 122 offenbart ist, erkenne ich gemäss Absatz 37, Bundesgesetzbuch, Paragraph 1.56(a) meine Pflicht zur Offenbarung von Informationen an, die zwischen dem Anmeldedatum der früheren Anmeldung und dem nationalen oder PCT internationalen Anmeldedatum dieser Anmeldung bekannt geworden sind.

I hereby claim the benefit under Title 35. United States Code. §120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, §122, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, §1.56(a) which occured between the filing date of the prior application and the national or PCT international filing date of this application.

PCT/DE00/01001 (Application Serial No.) (Anmeldeseriennummer)	03.04.2000 (Filing Date D, M, Y) (Anmeldedatum T, M, J)	anhängig (Status) (patentiert, anhängig, aufgegeben)	pending (Status) (patented, pending, abandoned)
(Application Serial No.) (Anmeldeseriennummer)	(Filing Date D,M,Y) (Anmeldedatum T, M; J)	(Status) (patentiert, anhängig, aufgeben)	(Status) (patented, pending, abandoned)

Ich erkläre hiermit, dass alle von mir in der vorliegenden Erklärung gemachten Angaben nach meinem besten Wissen und Gewissen der vollen Wahrheit entsprechen, und dass ich diese eidesstattliche Erklärung in Kenntnis dessen abgebe, dass wissentlich und vorsätzlich falsche Angaben gemäss Paragraph 1001, Absatz 18 der Zivilprozessordnung der Vereinigten Staaten von Amerika mit Geldstrafe belegt und/oder Gefängnis bestraft werden koennen, und dass derartig wissentlich und vorsätzlich falsche Angaben die Gültigkeit der vorliegenden Patentanmeldung oder eines darauf erteilten Patentes gefährden können.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

German Language Declaration

VERTRETUNGSVOLLMACHT: Als benannter Erfinder beauftrage ich hiermit den nachstehend benannten Patentanwalt (oder die nachstehend benannten Patentanwälte) und/oder Patent-Agenten mit der Verfolgung der vorliegenden Patentanmeldung sowie mit der Abwicklung aller damit verbundenen Geschäfte vor dem Patent- und Warenzeichenamt: *(Name und Registrationsnummer anführen)*

POWER OF ATTORNEY: As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) to prosecute this application and transact all business in the Patent and Trademark Office connected therewith. *(list name and registration number)*

And I hereby appoint

Customer No. 21171

Telefongespräche bitte richten an: *(Name und Telefonnummer)*

Direct Telephone Calls to: *(name and telephone number)*

Ext. _____

Postanschrift:

Send Correspondence to:

Staas & Halsey LLP
700 Eleventh Street NW, Suite 500 20001 Washington, DC
Telephone: (001) 202 434 1500 and Facsimile (001) 202 434 1501
or
Customer No. 21171

Voller Name des einzigen oder ursprünglichen Erfinders:	Full name of sole or first inventor:
Dr. PETER LIGGESMEYER	Dr. PETER LIGGESMEYER
Unterschrift des Erfinders Datum: 8.11.2001	Inventor's signature Date
Wohnsitz: Potsdam, DEUTSCHLAND DE	Residence: Potsdam, GERMANY
Staatsangehörigkeit: DE	Citizenship: DE
Postanschrift: Eichenring 27, 14469 Potsdam	Post Office Addess: Eichenring 27, 14469 Potsdam
Voller Name des zweiten Miterfinders (falls zutreffend):	Full name of second joint inventor, if any:
Unterschrift des Erfinders Datum	Second Inventor's signature Date
Wohnsitz ,	Residence ,
Staatsangehörigkeit	Citizenship
Postanschrift	Post Office Address

(Bitte entsprechende Informationen und Unterschriften im Falle von dritten und weiteren Miterfindern angeben).

(Supply similar information and signature for third and subsequent joint inventors).

Form PTO-FB-240 (8-83) Patent and Trademark Office-U.S. Department of COMMERCE

Glossary of variable names and file names

AS_GraphKante.h	AS_GraphEdge.h
GraphKanteC	GraphEdgeC
Kante	Edge
KantenTyp	EdgeType
KantenTypT	EdgeTypeT
SchleifenEnde	LoopEnd
Nachfolger	Successor
Eckentyp	Corner_type
Ecke	Corner
Graphtyp	Graph_type
gerichtet	directional
istGerichtet	isDirectional
AnzahlKanten	NumberEdges
verbindeEcken	connectCorners
ZyklusUndZusammenhang	CycleAndConnection
wert	value
Kanten	Edges
Komponentenanzahl	Number_of_components
nichtBesucht	notVisited
EckStatus	CornerStatus
besucht	visited
bearbeitet	processed
Eckenzustand	Corner_state
EckenStack	CornerStack
dieEcke	theCorner
Nachf	Succ
Zyklen	Cycles
Ziel	Target
Kantenwert	Edge_value
GetGatterIdent	GetGateID
Anzahl_Defs	Number_Defs
Anzahl_Uses	Number_Uses
Anzahl_P_Uses	Number_P_Uses
Anzahl_Deklarationen	Number_Declarations
Schleifenentscheidungen	Loop_decisions
Entscheidungen	Decisions

Zaehlschleifenentscheidungen	count_loop_decisions
KnotenNummern	Node_Numbers
KnotenIdentifizierer	Node_Identifiers
ListeAusgeben	outputList
zaehleDeklarationen	countDeclarations
basisgroessenInDatei	basicvariablesInFile
KFGListeAusgeben	outputCFGList
KFGListeT	CFGListT
KnotenIdentT	NodeIDT
Nr	No
KnotenIdent	NodeID
KnotenNummer	NodeNumber
KFGListeC	CFGListC
uwgknoten.h	uwgnode.h
uwgkante.h	uwgedge.h
sliceAusgeben	outputSlice
posGatter	posGate
uwgkanteC	uwgedgeC
uwgknotnC	uwgnodeC
maxKnotentextLength	maxNodetextLength
maxBemerkLength	maxCommentLength
maxWahrVarLength	maxTrueVarLength
ftgatter	ftgate
gattertext	gatetext
gatterbemerkung	gatecomment
eingefügterKnotenFla	insertedNodeFla
EingefügterKnoten	InsertedNode
KnotenNummerEnde	NodeNumberEnd
SetKFGKnotenNummer	SetCFGNodeNumber
Zahl	Number
Zaehler	Counter
Verzweigung	Branch
Bemerkung	Comment
Gatterbemerkung	Gate comment